Report No. 257

COO-1469-

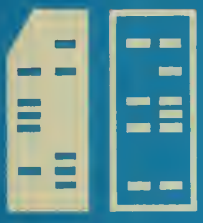


# GRAPHICAL SPECIFICATION OF COMPUTATION

by

Fontaine K. Richardson

April 10, 1968



DEPARTMENT OF COMPUTER SCIENCE · UNIVERSITY OF ILLINOIS · URBANA, ILLINOIS

COO-1469-

Report No. 257

# GRAPHICAL SPECIFICATION OF COMPUTATION*

by

Fontaine K. Richardson

April 10, 1968

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

* This work was supported in part by Contract No. US AEC AT(11-1) 1469 and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, June, 1968.

# GRAPHICAL SPECIFICATION OF COMPUTATION

Fontaine Kelleam Richardson, Ph.D.
Department of Computer Science
University of Illinois, 1968

## ABSTRACT

This thesis reports on an application of on-line computer graphics to the problem of computer programming. In this investigation, a class of computer systems is proposed which reduces the cost of on-line computer graphics. One of these systems consists of a free standing, inexpensive display device (CRT, light pen, and small computer) connected via a communication link (phone line) to a large supporting computer. The display device is used for the preparation of graphical images (computer programs) and the supporting computer is used for the manipulation of these graphical images (executing the computer programs). The cost of on-line computer graphics is reduced because the display device is inexpensive (projected costs start at $10,000) and the supporting computer is used under the control of a time sharing system which allows the supporting computer to be used for other tasks while it is not needed by the display device. An experimental system, fitting this description, was assembled for this study.

The graphical computer programs take the form of flowcharts whose symbols contain executable statements in a "higher level" language. The symbols are connected by arrows that indicate the sequence in which the symbols are to be executed. A particular "high level" language, FPL/I, has been implemented for this investigation. A statement in FPL/I

consists of concatenated references to primitives ("machine instructions") and flowcharts. The arguments for the primitives and the flowcharts are written in a prefix notation. FPL/I executes on a list oriented machine that is simulated by the supporting computer. FPL/I operates in an environment that provides debugging aids such as 1. tracing a flowchart's execution; 2. suspending, resuming, or terminating a flowchart's execution; and 3. examining or altering the value of an identifier.

Based upon the experimental system constructed for this investigation, several observations are made and possible changes noted.

# ACKNOWLEDGMENT

The author wishes to express his gratitude to Professor C. W. Gear for his constant encouragement, indispensable guidance, and constructive criticisms in the preparation of this thesis.

The author is also indebted to the Department of Computer Science, University of Illinois, for its support, Tang-Yung Lo for his extensive work in programming the flowchart editing system, and Marguerite Dunlap for her typing of this thesis.

Finally, the author extends his appreciation to his wife, Judy, without whose support this undertaking would presumably still have been possible, but who surely hastened its conclusion.

TABLE OF CONTENTS

CHAPTER Page

# LIST OF FIGURES

# CHAPTER I

# INTRODUCTION

This is an investigation of flowchart programming languages. Our interest is in allowing a programmer to work with a computer program as a flowchart rather than as a card deck. Figure 1 is a flowchart program, SQRT, that calculates the square root of a number. SQRT is written in an experimental language, FPL/I, implemented as part of this investigation. Briefly, FPL/I instructions are composed of "machine instructions" and invocations of flowcharts. The arguments for the instructions and flowcharts are written in prefix notation. Floating point numbers are the data for FPL/I.

SQRT performs its calculation iteratively. At symbols (2)-(4), SQRT accepts the input parameter, creates identifiers for temporary use, and stores an initial approximation of the square root of X in Y (Y=(1+X)/2). At symbol (5), TEMP is computed (TEMP=(X/Y-Y)/2) both as a measure of how close Y is to the square root of X and as a means of improving Y's approximation. At symbol (6), a decision is made. If Y is not "close enough" (TEMP < 0) then control goes to symbol (7) where Y is improved as an approximation and then control goes back to symbol (5). If Y is "close enough" (TEMP > or = 0), the control goes to symbol (8) where the temporary identifiers are released and the square root is returned to the "calling" flowchart.

The reasons for considering flowcharts as programming media are: 1. their long use in the planning, coding, and documenting phases of the "programming process" and 2. the advent of hardware that provides inexpensive flowchart input and output facilities. The work of Ellis and Sibley on RAND Corporation's GRAIL project[9] and of W. R. Sutherland

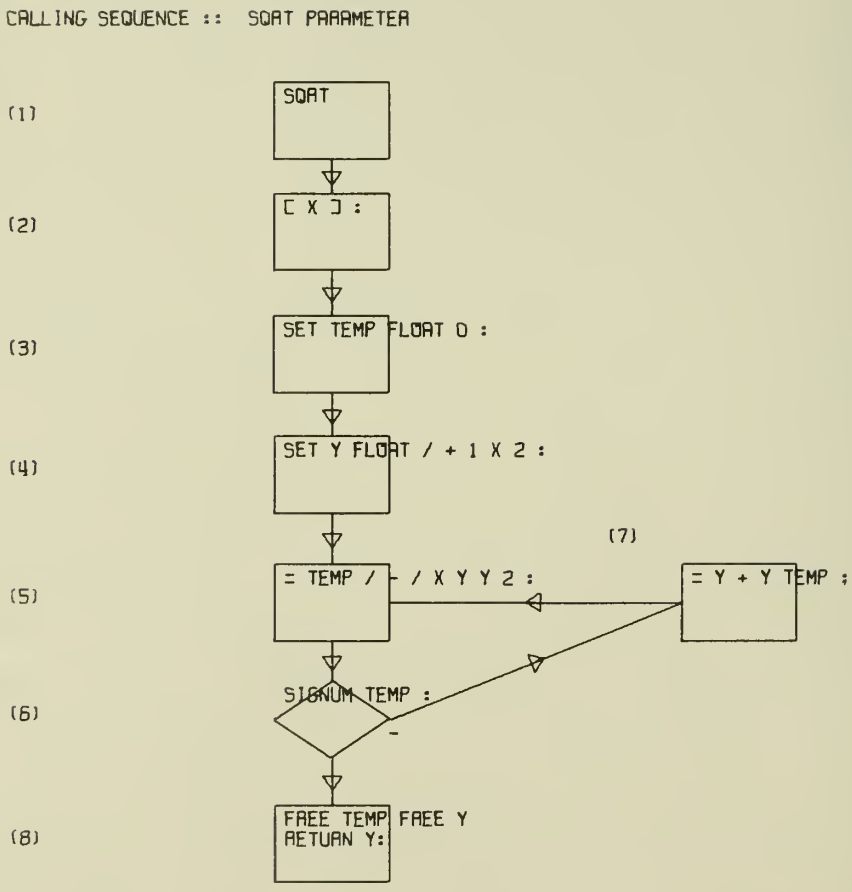



Figure 1. A SQRT Flowchart in FPL/I

on the TX-2 at Lincoln Laboratories, MIT[22] indicates that flowchart programs are valuable tools which deserve more attention and wider application. These investigations use light pens and Cathode Ray Tubes (CRT) for flowchart Input/Output (I/O).

Our purpose in this investigation is to examine flowchart programming languages in relation to algebraic programming languages and to experiment with the use of flowcharts as programming tools.

# CHAPTER II

# FLOWCHARTS

## 2.1 Why Flowchart Programming Languages?

The mention of a flowchart calls to mind 1. the sketch that is used to conceive and plan a computer program, 2. the detailed and carefully drawn flowchart that is used as an aid in the coding of the program, and 3. the finished flowchart that represents part of the documentation of the program. These three types of flowcharts are different in function, form, and detail. Let us see examples of these types of flowcharts as they are used in the development of a square root routine. The flowchart of Figure 2 represents the analysis of the process required to compute the square root of a number, the flowchart of Figure 3 represents the detailed flowchart used in the coding of the FORTRAN SQRT subroutine of Figure 4, and Figure 5 represents part of the final documentation of the square root routine.

These flowcharts are valuable tools for the programmer. However, the flowcharts are incomplete in the sense that a computer program is written in a conventional programming language that is not oriented towards flowcharts. Existing languages were designed within the framework that current computer I/O consists of character strings or lines. Thus, a program is input into the computer as a set of punched card images and is not input as a flowchart. Therefore, a basic difficulty encountered in trying to use flowcharts is the required conversions between flowchart and program. For example, a detailed flowchart would have to be converted into a program statement in order to execute the program. The debugged program would have to be converted back into a flowchart for program documentation.

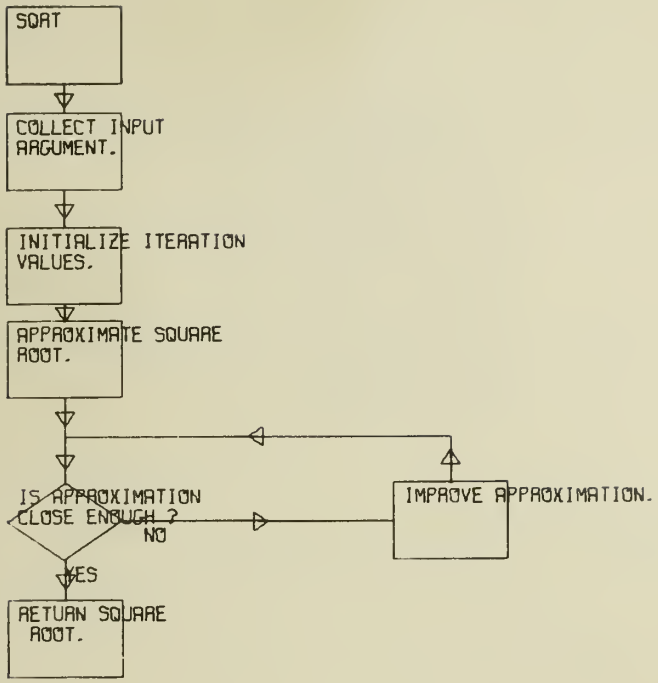



Figure 2. A Planning Flowchart for a Square Root Routine

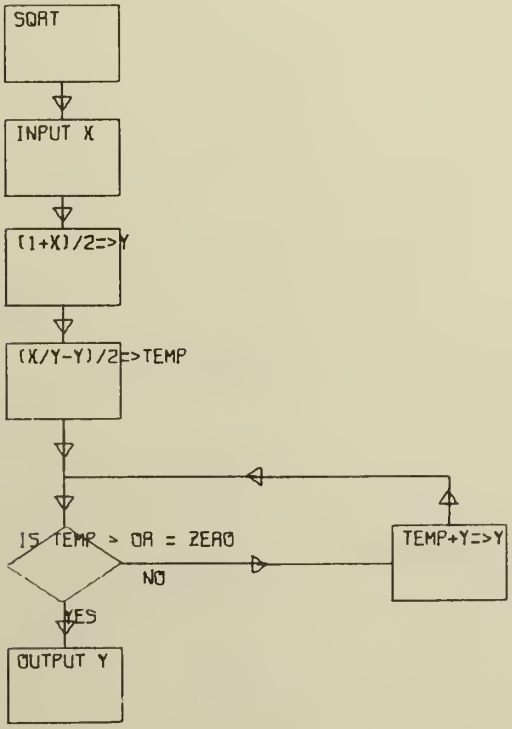



Figure 3. Detailed Flowchart for Square Root Routine

```
(1)       SUBROUTINE SQRT (X,Y)
     C
(2)       Y=(1+X)/2
     C
(3)    1  TEMP=(X/Y-Y)2
     C
(4)       IF (TEMP) 2,3,3
     C
(5)    2  Y=Y+TEMP
     C
          GO TO 1
     C
(6)    3  RETURN
     C
          END
```

Figure 4. FORTRAN Square Root Subroutine

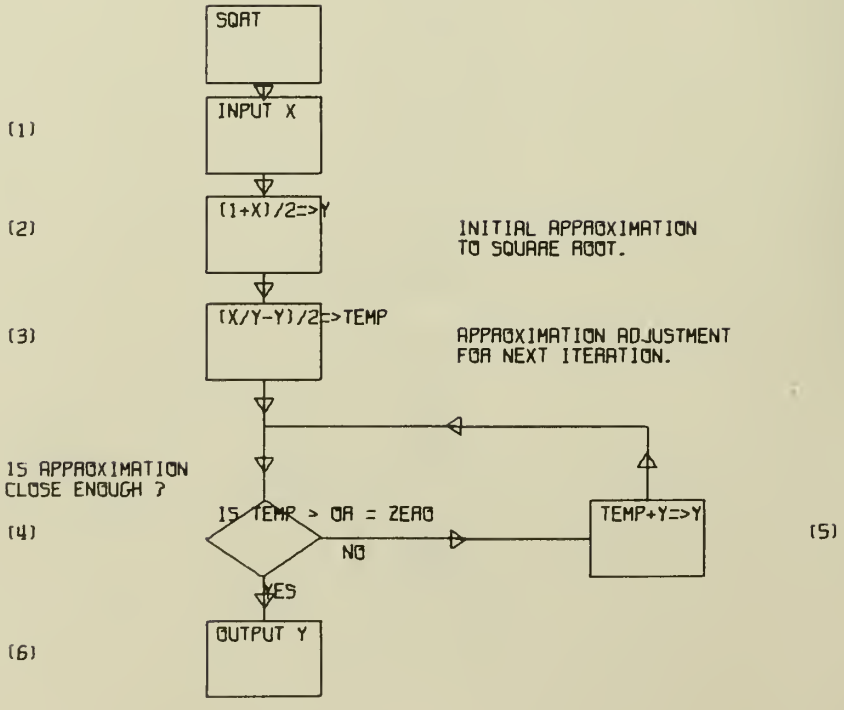



Figure 5. Documentation Flowchart for Square Root Routine

The use of flowcharts requires the provision of direct computer I/O of flowcharts and for the execution of flowchart programs. In order to specify flowchart programs, there must be a flowchart programming language.

## 2.2 Existing Flowchart I/O Methods

Line printers, incremental plotters, and CRTs all have been used to generate flowchart output. The flowcharts that are produced quickly and easily on a line printer are difficult to use because of the line printer's lack of graphical resolution. The flowcharts generated with an incremental plotter are of excellent graphical quality but the plotter turnaround time is large. The flowcharts quickly displayed on a CRT are of excellent graphical quality. These flowcharts are directly viewed by the user or are automatically photographed. The direct viewing of a displayed flowchart is expensive. The cost is in the equipment necessary to generate the display. The cost of automatically generating photographs of the CRT is directly proportional to the speed with which these photographs are taken and developed. Electrostatic printers, a special form of CRT photography, are fast but expensive. Flowchart output is obtainable with only moderate expense. However, flowchart input is more expensive.

A graphical image is not amenable for computer input. The computer input form of a graphical image is its digitization. Therefore, flowcharts must be digitized for their input to the computer. One method of digitization is with the use of a light pen attached to a computer that has a CRT display. The computer generates a point display on the CRT, the light pen "sees" a point on the CRT, and the computer determines the pen's coordinates. As the user moves the pen, the

computer maintains a record of the pen's coordinates. These internally held coordinates form a digital image of the externally traced graphical information.

Another digitizing scheme is the drafting table and stylus. As the user moves the stylus on the table, the stylus coordinates are generated by mechanical linkages to the stylus. These coordinates are written on magnetic tape, punched into cards, or transmitted to a computer. The first two methods of inputting the stylus' coordinates into a computer have the disadvantage that they provide no "feedback" of the digitized information. This "feedback" is essential for allowing the alteration of existing digitizations. Another form of the drafting table and stylus is the RAND tablet and stylus connected to a computer which has CRT display. The stylus coordinates are generated electronically rather than mechanically. The CRT displays the "feedback" of the digitized information. This "feedback" allows the user to alter existing digitizations.

The digitizing process can be automated by using a television camera. However, the organization and volume of the information produced by the camera is different from that generated by the light pen or RAND tablet. The camera generates large coordinate matrices of intensity levels while the pen or tablet produces the end points of straight lines. The cost of any of these schemes is in the generation and processing of the digitization. This processing is required in order to recognize the input symbols and text for flowchart input.

A modified digitizing scheme for flowchart input is to digitize only the non-texual information on a flowchart and maintain the text as characters on a punched card. This scheme eliminates most of the

processing but still requires a large amount of digitization. Another version of this scheme is to digitize the coordinates of a flowchart symbol's center, represent the symbol type with a name, and maintain the text as characters on cards. In this scheme, little digitization is done and all of the flowchart's information is represented as characters on a card. The flowchart input is the user generated card version of his flowchart. The value of this approach is that cards and card images are a common form of computer I/O. The disadvantage of this approach is that the work required to input flowcharts is done by the user and not done by a computer.

## 2.3 Inexpensive Flowchart I/O

Previous investigations of flowchart I/O have involved expensive computers maintaining CRT displays. Typically, the CRT display is attached to the computer as a high speed I/O device and the computer is dedicated to the task of display maintenance. These arrangements are not practical for extended use of the flowchart I/O facilities.

In recent years, two developments have occured that make possible practical flowchart I/O. These developments are: 1. the availability of small, inexpensive computers and 2. the availability of time sharing systems on large, high speed computers.

Several different computers are commercially available that are small, general purpose, and inexpensive. A light pen, a RAND tablet, and a CRT are economically attached to any one of these computers. The computer's memory is used for program and display storage. The computer's arithmetic capability is used to manipulate the display information and to process the input from the pen, tablet, or teletype. Any one of

these hardware configurations is a display device. Using a display device, flowcharts can be constructed by the user on the CRT.

In any of these display devices, the arithmetic capability and storage capacity are limited. A time-sharing system on a large high-speed computer has arithmetic capability and memory capacity that is available on a demand basis. If the display device is established as a user of the time-sharing system, then problems are sent to the large computer for solution when the display device's facilities are overtaxed. Thus the large computer is the supporting computer for the display device. For example, the display device's facilities are quickly overtaxed for the purpose of executing or storing several flowchart programs that have been constructed at the display device. These programs are sent to the supporting computer for execution or storage. In this capacity, the small computer is a flowchart I/O device for the supporting computer.

The connection of the display device to the supporting computer is an economic method for flowchart I/O. The supporting computer expense is directly related to its use.

# CHAPTER III

# FLOWCHART USE

In widespread usage, flowcharts have been used to plan computer programs and document existing programs. In specific investigations, flowcharts have been used to represent executable programs.

## 3.1 Computer Drawn Flowcharts

The construction and editing of flowcharts is a time consuming and tedious task. For this reason, there have been attempts to automate the production of these flowcharts. One automation effort is a program that produces flowcharts on a printer, plotter, or CRT. The input to the program is a card deck representing the flowchart. The card input is generated from a flowchart hand drawn on a prepared form which is preprinted with a set of coordinates. The input deck contains each symbol's page number, page coordinates, shape specification, and text. In this way, the input controls the flowchart's layout and content.

The difficulties encountered in using this scheme are producing the original card representation and altering the cards to reflect changes in the flowchart. To make alterations to the flowchart, the existing representation is changed or a new one is generated. This scheme does generate computer drawn flowcharts. The turnaround time required to produce one of these flowcharts is long. Therefore, this scheme is usually only used to produce flowcharts for documentation.

## 3.2 Automatic Flowchart Generation

Another flowchart automation effort is a system whose input is a program's source representation and whose output is the program's flowchart produced on a printer, plotter, or CRT. For example, suppose

that the system's input is a card deck for the FORTRAN program Z. The system's output is program Z's flowchart. Each symbol on the flowchart contains a sequence of the program's statements. Different symbol shapes contain different statement types. For example, a diamond shaped symbol contains a conditional statement, a rectangle contains arithmetic statements, and a circle contains an off-page reference. The arrows connecting these symbols are derived from the program's labeled, conditional and transfer statements. In this way, a direct correspondence is established between the input source program and the generated flowchart. An example of a program and a possible form for its generated flowchart is shown in Figures 6 and 7.

These flowcharts are generated quickly and easily because the system's input is the same as that used to compile and execute the program. Therefore, no extra human effort is required to generate a program's flowchart.

The direct correspondence between a program and its flowchart makes the flowcharts valuable as debugging tools and as part of a program's documentation. In debugging a program, a flowchart is generated for each program change. In this manner, the flowchart "is" the program and the flowchart instead of the program listing is used for debugging. The value of the flowchart as documentation is directly related to the value of the program listing as documentation. For example, a FORTRAN program's listing is more valuable as documentation than an assembly language program's listing. The same statement is true of the program's automatically generated flowcharts.

The existence of this system is due to the lack of flowchart input facilities, and the need for flowchart generation. The

```
      :
      A=B+C
C
      IF (A) 3,4,5
C
   3  D=F+G*H
C
   4  J=K+L
C
      GO TO 6
C
   5  M=N
C
      GO TO 3
C
   6  O=P*Q
      :
```

Figure 6. Segment of a FORTRAN Program Z

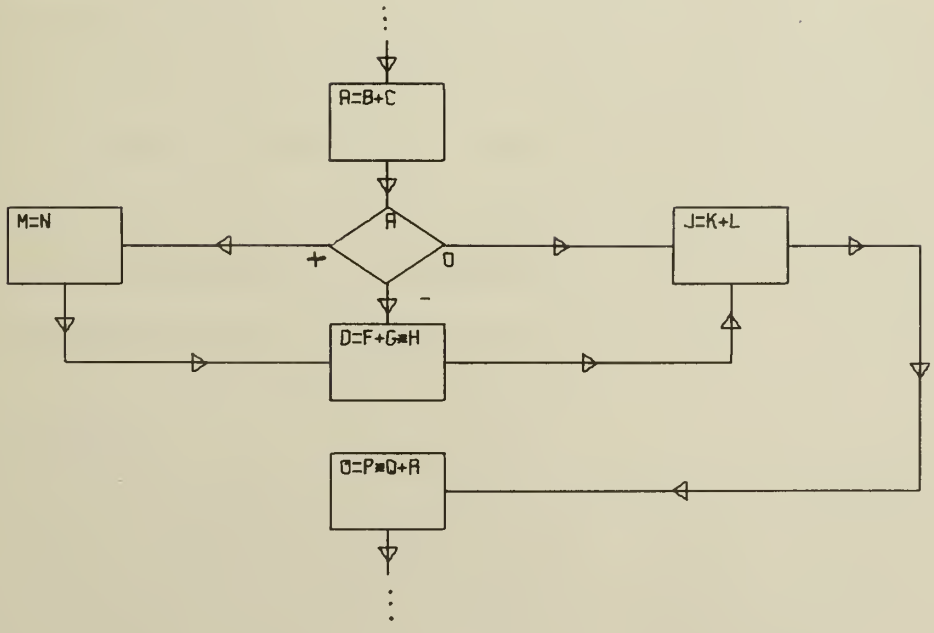



Figure 7. Segment of the Automatically Generated Flowchart for Z

disadvantages of this system are due to the automatic flowchart layout. For example, the appearance of the flowcharts for a program and its successor can vary drastically with what are apparently small program changes. Thus in using the flowchart as a coding or debugging tool, the programmer must frequently readjust to an altogether different flowchart of almost the same program. This periodic readjustment on the part of the programmer can cause these flowcharts to fall into disuse. This is because a programmer develops a "feel" for the way certain flowchart sections "look" and uses this information in rapidly scanning a flowchart to find a specific section. This is analogous to the way a person reading a book will remember the position and format of an item on a page and will use this information to locate the item.

Another problem associated with automatic flowchart layout is the clear presentation of the program. For example, it is difficult to automatically layout an easily read flowchart for a program that contains a large number of branches. The problems of automatic layout are solved by allowing the programmer to do the layout. For this, the programmer must be able to input flowcharts or manipulate flowcharts within the computer. The expense of this solution has been a limiting factor.

In practice, the problem of analyzing the source and drawing a reasonable form of the flowchart is difficult, and has not been adequately solved. Knuth[17] requires that the user do the analysis, and the resulting flowchart is "linear", being a one for one map of the input. Haibt[11] states that an attempt is made to analyze the program flow, but gives no algorithms. Neither does he indicate how he will layout the resulting flowcharts. Hain and Hain[12] tackle the layout problem only, arriving at a layout by removing enough paths to cut all

program loops--one of the most important parts of the program.

Systems for the automatic flowchart generation exist for particular languages--FORTRAN and COBOL. FLOWTRACE[20] is an interesting extension of these systems.

FLOWTRACE requires an initial input of a language L's description. Then for the input of a program written in L, FLOWTRACE generates the program's flowchart. Thus FLOWTRACE generates a flowchart for a program written in any "describable" language.

The required language description is not complex because FLOWTRACE is concerned only with the syntax of statement labels, statement type, and transfer statements. Therefore, new languages are accommodated by these easily constructed language descriptions. Also, these new languages do not need to be programming languages. For example, a new language can be constructed from FORTRAN by using only the comment statements and control transfer statements. The generated flowchart contains only the comments and branch parts of the FORTRAN program. This flowchart is less detailed than the program listing and is valuable for documentation. FLOWTRACE does not, however, handle block structured languages such as ALGOL[18] and PL/I[15].

### 3.3 The GRAIL Project

In T. O. Ellis and W. L. Sibley's work on RAND Corporation's GRAIL Project [9] the emphasis is on developing the CRT as a common working surface between the human and the computer. The effort is directed toward creating a software-hardware system in which the human manipulates the contents of the CRT directly and naturally, without explicit reference to the computer. The RAND computer system is a large computer (IBM 360/40) attached through an I/O channel to a CRT

and a RAND tablet.

For the study of human's CRT use, Ellis and Sibley have specified a flowchart programming language and implemented a supporting system. The flowchart symbols are drawn with the RAND tablet's stylus and the drawing is displayed on the CRT. The standard representation of the hand-drawn symbol is displayed if the completed drawing is recognizable. The flowchart programs are composed of symbols, interconnecting lines, and the text on each symbol. The text within a symbol conveys its semantic content. The text represents program statements in assembly language. These flowchart programs may be edited, filed within the computer system, executed or debugged within the system.

## 3.4 W. R. Sutherland's Work

W. R. Sutherland discusses graphical programming and demonstrates an experimental graphical programming system in his thesis [22] work at Lincoln Laboratories, MIT. This programming system operates in the environment of the CRTs and RAND tablets attached via an I/O channel to the TX-2. Flowchart program construction, editing, storage, execution, and debugging are possible within this programming system.

The method of constructing flowcharts is derived from I. E. Sutherland's SKETCHPAD [21]. For example, from SKETCHPAD the DRAW command is used to construct symbol definitions and connections, the picture creation facility is used to define symbols, and the picture reference facility is used to create symbol instances. The recursive subpicture facility is used to define a flowchart as a symbol. This allows a flowchart to be referenced by use of its symbol.

The relation of a flowchart program to its execution is a basic part of W. R. Sutherland's work. For this, the convention used is

that a flowchart's semantics are provided by the shapes and interconnections of its symbols. A symbol is an operator that is activated by its inputs and that produces outputs. The shape of the symbol determines the operation performed on the inputs. The lines connecting symbols on the flowchart represent data. A line connected to a symbol's input terminal is an input datum for that symbol. A line connected to a symbol's output terminal represents an output datum from that symbol. The splitting of a datum line means that the datum represented by the line is an input for more than one symbol. This splitting of data lines is the notation used for indicating possible parallelism in a data flowchart program. The convention is that each symbol performs its operation when its inputs are defined. Thus if two or more symbols have their inputs defined simultaneously, then these operations are to be done in parallel. This type of flowchart is referred to in this thesis as a data flowchart because its data are represented as lines. These data lines are analogous to wires connecting operational elements (flip flops, NAND gates, etc.) in an electronic circuit. These data flowcharts might also be called analog because of their similarity to circuit diagrams.

These data flowcharts are in contrast to the SQRT flowchart of Figure 1 where the data are represented by conventional identifiers and the symbol connections indicate control flow. Flowcharts similar to the SQRT flowchart are referred to as control flowcharts. Figure 8 is a data flowchart of Sutherland's that iteratively computes the square root of its input. In the flowchart, the lines connected to the tall side of each symbol are inputs and the lines connected to the short side are outputs. A symbol with a small circle on one of its inputs

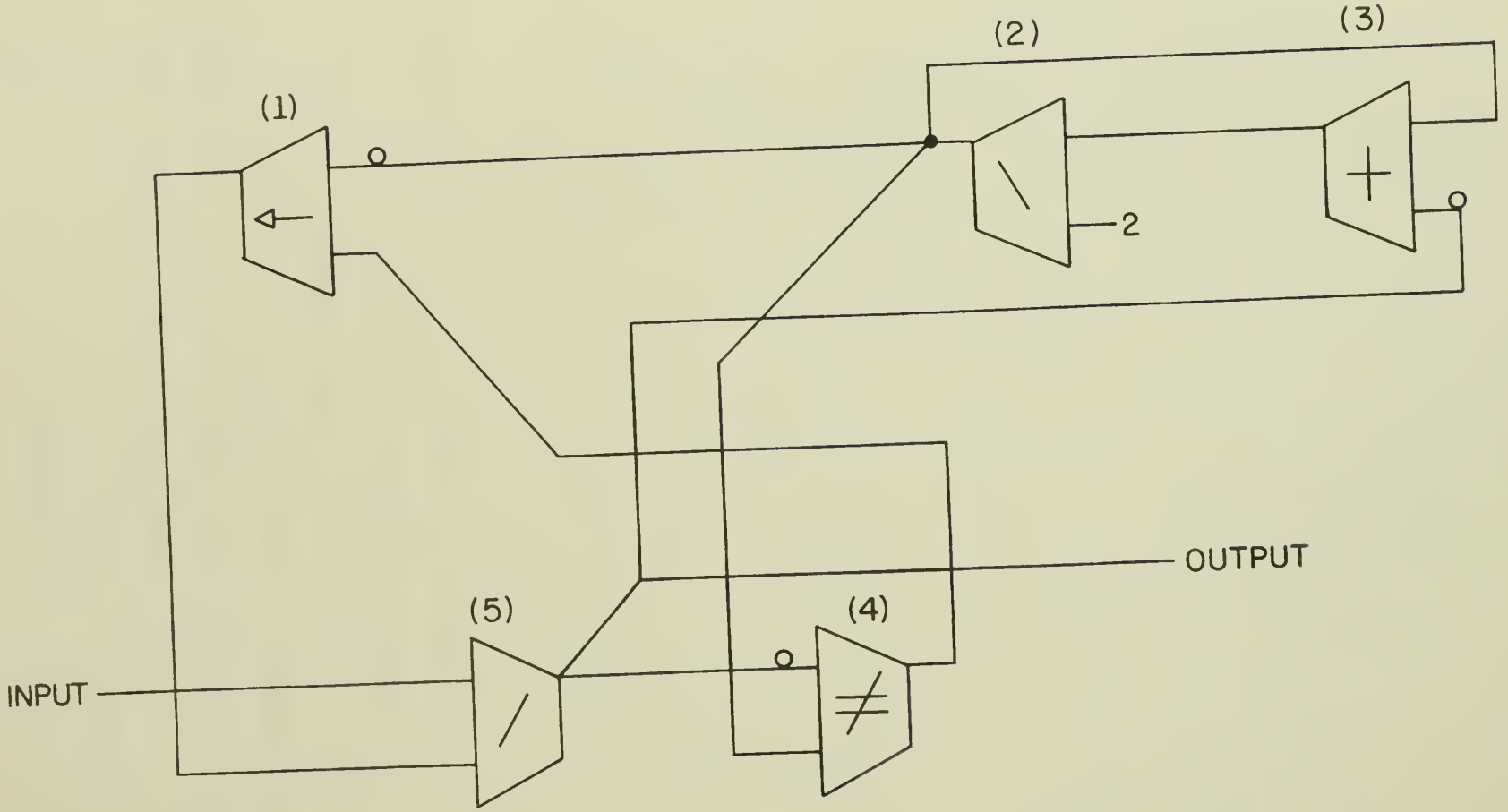



FIGURE 8
DATA FLOWCHART FOR SQUARE ROOT

is only activated for each new value appearing on that input connection. The other operators are continuously activated. The flowchart's inputs are data lines that are not outputs from symbols on the flowchart. The flowchart's outputs are data lines that are not used as inputs to symbols on the flowchart.

In Figure 8, the division operators (2) and (5) output the quotient of the "top" input and the "bottom" input with the "top" input as the quotient's numerator. The numeral 2 on the "bottom" input of (2) is a constant input of the number 2. The addition operation (3) outputs the sum of the two inputs. The unequal operator (4) outputs "true" or "false" if the two inputs are or are not unequal. The "pass through" operator (1) outputs the "top" input if the bottom input is "true". This operator at (1), with its circled input, is a method of controlling the flowchart's execution.

The data flowchart for square root uses the same algorithm that the SQRT flowchart of Figure 1 does. The differences between data and control flowcharts are indicated by the differences between these two square root routines.

# CHAPTER IV

# COMPUTER AIDED PROGRAMMING SYSTEM

The Computer Aided Programming System (CAPS) is provided as a tool for use in the "computer programming process". CAPS is not limited to a particular computer or language but is applicable to a class of computer systems and programming languages.

CAPS permits a user to plan, code, debug, document, and alter computer programs expressed in the media of flowcharts.

## 4.1 Computer Systems

A CAPS computer system consists of a display device connected via a communications link to a supporting computer. The display device provides flowchart I/O between the user and the supporting computer. The supporting computer manipulates these flowcharts at the user's command.

An experimental CAPS computer system has been assembled and used in this investigation. The display device is a Programmed Buffered Display type 338, the supporting computer is the ILLIAC II, and the communication link is direct connection, similar to a leased phone line, that has a speed of 100 characters per second. (See Appendix A for a more detailed description of this experimental system.)

## 4.2 Flowchart Input

The user inputs a flowchart at the display device with the light pen and keyboard. The display device monitors these inputs and constructs an internal flowchart representation based upon their activity. This internal representation is used to create a flowchart display on the CRT. The supporting computer receives the flowchart's

representation piecewise as it is constructed or as a complete unit after it is constructed.

In CAPS, a flowchart is defined as all of the flowchart information in the display device's memory at any one time. PROCESS, DECISION, ARROW, CONNECTOR, and TEXT are the allowed symbol types. (See Figure 9 for their displayed forms.) A symbol is an instance of one of these symbol types. All instances of the same symbol type have identical shape, size, and orientation, but they do not necessarily contain the same textual information.

Relative to a flowchart, the basic set of drawing operations are:

1. add a symbol,
2. delete a symbol,
3. edit text onto a symbol, and
4. move a symbol's position.

A particular operation is activated by light penning its light button. Once an operation is specified, displayed visual cues indicate when an operand is to be light penned. All of these basic operations are done using the light pen.

In the experimental system, a symbol is moved with the light pen after the "move" operation is specifed. Upon selecting a symbol, the pen's position is indicated by a tracking cross. The light pen, tracking cross, and symbol now move in unison. The symbol's position is fixed by closing the light pen's shutter. For a special case of the "move" operation, a symbol of each type is displayed in a special position on the CRT. These are symbol light buttons and are used for adding new symbols to the flowchart. A symbol is added to the flowchart

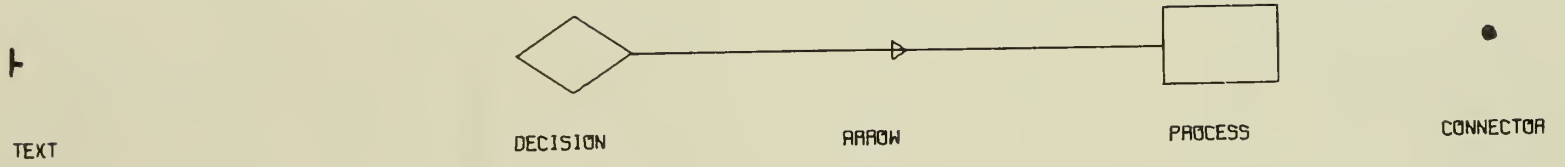



Figure 9. Allowed Symbol Types in CAPS

by "moving" a symbol light button. A copy of the symbol light button is "moved" with the light pen and the symbol light button's position remains fixed. This new symbol is now part of the flowchart. An ARROW symbol is added to the flowchart differently because an ARROW goes from one symbol to another symbol. To add an ARROW, the "move" operation is turned on and the ARROW symbol light button is "moved". Now, the word "FROM" is displayed as a visual cue indicating that the new ARROW will come from the next symbol light penned. After the "from" symbol is selected, the word "TO" is displayed indicating that the new ARROW will go to the next symbol light penned. The ARROW is displayed after the "TO" symbol is selected. An ARROW moves when either its "to" or "from" symbol is moved. These methods of adding symbols to the flowchart do not require the display device to do pattern recognition on arbitrarily drawn symbol shapes. Any flowchart can be constructed using these basic operations. (For a detailed description of the experimental drawing operations see Appendix B.)

## 4.3 Flowchart Representation

The data structure for flowchart representations combines the use of lists and tables. For a flowchart's representation, two tables are used. The table names are INFORM and LOCATE. INFORM contains all the flowchart information and occupies all of the memory not used for other purposes. LOCATE facilitates the addressing, garbage collection, and compacting of INFORM. LOCATE has length n, where n is determined for each implementation.

INFORM has a variable length entry for each symbol. An entry contains a symbol's type designation, X-Y coordinates, and text. Each symbol is assigned a number according to the order of its entry's

appearance in INFORM. For example, the M + 1st entry in INFORM is the symbol numbered M.

Symbol numbers are used by the ARROW entries in INFORM. An ARROW symbol goes to a symbol from another symbol rather than having position coordinates X and Y. Each ARROW entry in INFORM contains the "from" and "to" symbol numbers instead of coordinate information. (See Figure 10 for the format of each entry in INFORM.)

Each INFORM entry is transformed into display device instructions that generate its symbol display. The display instructions generated from INFORM create the entire flowchart display.

LOCATE has a fixed length entry for each symbol. The $I^{th}$ entry in LOCATE contains the address of the $I^{th}$ symbol's entry in INFORM. For example, LOCATE (7) + 3 is the address of the $3^{rd}$ word in the $7^{th}$ symbol's entry in INFORM. (See Figures 11, 12, and 13 for a flowchart, its INFORM, and its LOCATE representations.)

LOCATE is a tool used in manipulating INFORM and the display. For example, an ARROW symbol goes to a symbol from another symbol on the CRT. In INFORM an ARROW symbol's entry contains the "to" and "from" symbol numbers i and k. The coordinates of symbols i and k are needed in order to construct the ARROW symbol's display. LOCATE (i) and LOCATE (k) are the addresses of the entries in INFORM for the symbols numbered i and k. The symbols' coordinates are obtained relative to these addresses. For example in the experimental system, the Y and X coordinates of symbol i are LOCATE (i) + 2 and LOCATE (i) + 3, respectively.

An alternative to the use of LOCATE is to have the ARROW symbol entry in INFORM contain the coordinates of its endpoints. This scheme allows direct access to the coordinates for display purposes; but does

| | | |
|---|---|---|
| A | symbol type, intensity, group light pen enable/disable, and blink on/off | |
| A+1 | Flowchart symbol's Y coordinate or "TO" symbol number (ARROW type) | |
| A+2 | Flowchart symbol's X coordinate or "FROM" symbol number (ARROW type) | |
| A+3 | N - number of words of text in this symbol | |
| A+4 | First character | Second character |
| A+5 | Third character | Fourth character |
| | . | . |
| | . | . |
| | . | . |
| A+3+N | 2N-1 character | 2N character |

Location A:
- bits 0-3 type
- bits 4-5 scale
- bit 6 group
- bit 7 light pen enable/disable
- bit 8 blink on/off
- bits 9-11 intensity

Type:
- 0 - Last symbol in the table
- 1 - DECISION
- 2 - PROCESS
- 3 - CONNECTOR
- 4 - ARROW
- 5 - TEXT

Figure 10. INFORM's Entry Format

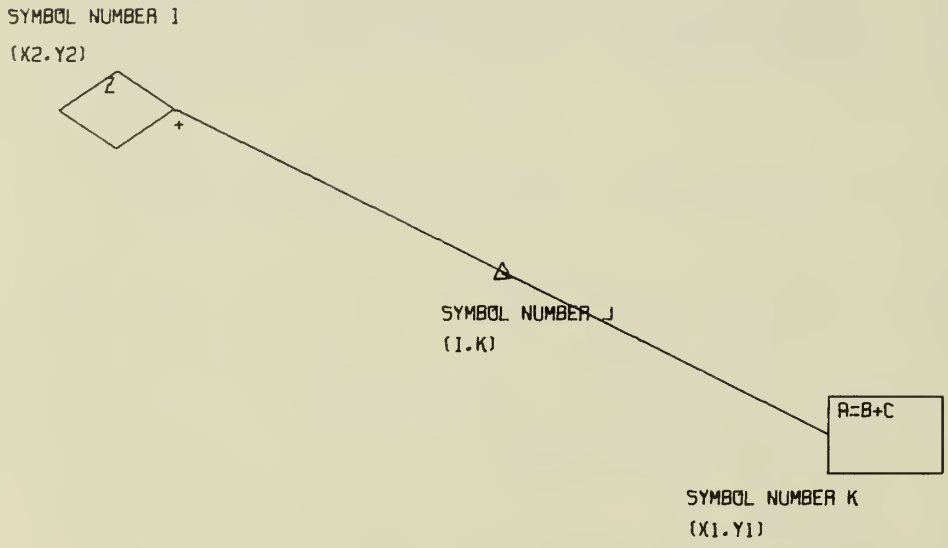



Figure 11. Segment of Flowchart A

IN EACH ENTRY:
SYMBOL TYPE, X COORDINATE, Y COORDINATE, TEXT OR
ARROW, "FROM" SYMBOL NUMBER, "TO" SYMBOL NUMBER, TEXT

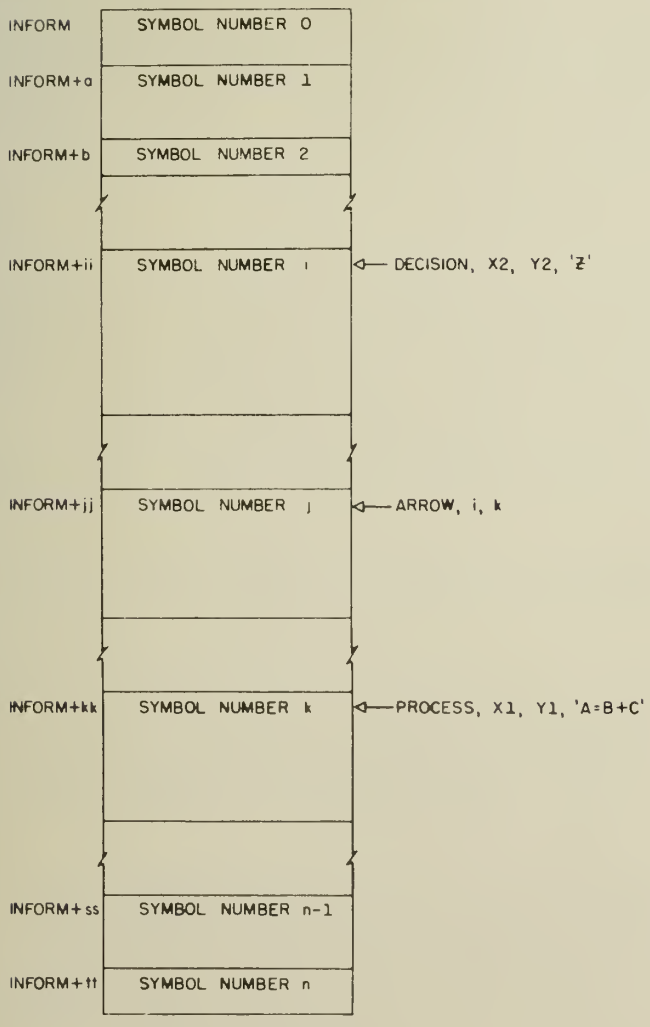



FIGURE 10b
INFORM FOR FLOWCHART A

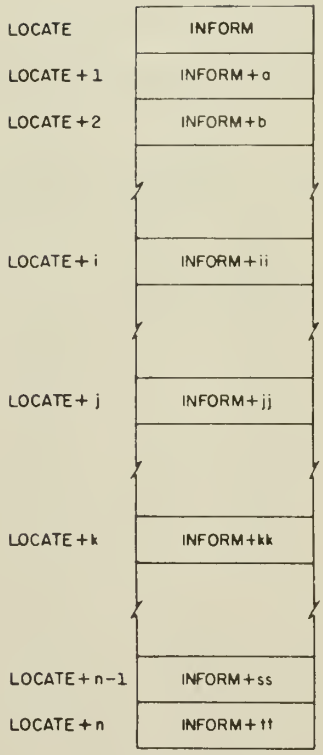



FIGURE 10c
LOCATE FOR FLOWCHART A

not allow a symbol's ARROWs to be "moved" as the symbol is moved. Another alternative is to have an ARROW symbol's entry contain the addresses of the INFORM entries for its "to" and "from" symbols. This scheme allows indirect access to the coordinates. A difficulty with this approach occurs in the compacting of INFORM. (This compacting is required after "garbage" collection on INFORM.) For compacting, each ARROW entry must have its "to" and "from" addresses altered. In the original scheme, LOCATE is recomputed after INFORM has been compacted. INFORM entries do not need to be changed.

## 4.4 Flowchart Input Extensions

For some users, the basic CAPS drawing operations (see section 4.2) are too primitive. For example, there are no operations that:

1. insure that two symbols are on a horizontal or vertical line,
2. manipulate - move, add, or delete - a group of symbols at one time,
3. change an existing symbol from one type to another,
4. allow the user to draw a flowchart that is larger than the CRT,
5. scale or selectively view this larger flowchart, and
6. allow the user to graphically define a new symbol type as a flowchart.

Operations 1. and 2. are "almost" accomplished using the basic operations. For example, to ensure that two symbols are on a horizontal or vertical line, "move" one of the symbols until the two symbols are on a horizontal or vertical line. To "move" a group of symbols, "move" the group one symbol at a time. Operations 3. through

6. are not possible with the basic operations.

Operations 1. through 6. and others are included in a specific CAPS implementation when memory space is available in the display device. Operations 1. through 5. are implemented in the experimental system. No changes are made to the data structure for the addition of a non-basic operation. This restriction is made in order to maintain compatibility of the data structures among display devices and supporting computers.

The way in which a non-basic operation is implemented is determined by the data structure. For example, in the experimental system, the operation "horizontal" places one symbol on a horizontal line through the center of another symbol. "Horizontal" accomplishes this by giving the Y coordinate of the first symbol the same value as the Y coordinate of the second symbol. However, there is no provision in the data structure which allows these two symbols to be permanently constrained to have the same Y coordinate. For example, after "horizontal" operates on two symbols, the "moving" of a symbol does not affect the position of the other symbol. This is in contrast to SKETCHPAD where the constraint feature of the data structure is used to ensure that two points are always on a horizontal line. The "moving" of one of these points causes the other point to "move" in order that both points continue to be on a horizontal line.

As another example from the experimental system, the "group" operation designates a set of symbols that are all manipulated at the same time. Any operation applied to a member of the "group" has an effect on all of the members. The "moving" of a member of the "group" causes all of the members of the group to move as if the group were a

rigid body. The "deleting" of a "group" member "deletes" all the "group" members. There is only one "group" and each symbol on the flowchart is either "in" or "out" of the "group".

The CAPS "group" operation is in contrast to the SKETCHPAD facility that allows a set of points, lines and constraints to be defined as a picture and manipulated - moved, rotated, scaled, or deleted - as a unit. Any number of pictures may be defined. A picture may appear within a picture which appears within a picture and so on. This definition facility is possible because of the SKETCHPAD ring structure which is similar to CORAL[19]. The CAPS "group" operation is similar to, but not as powerful as, the SKETCHPAD picture definition facility.

These non-basic operations are drawing tools provided by a display device. They manipulate the data structure and therefore the flowchart display. The effect of these operations on the data structure is limited because of the absence of the constraints and the generalized ring structure.

The question is: why not augment the CAPS data structure to allow constraints and generalized ring structures? The answer is found in the fact that flowchart representations must fit into the display device's memory and must be transmitted between the display device and the supporting computer. The more powerful data structures occupy more memory space and take a longer time to transmit over the communication link. The CAPS data structure was designed to represent flowcharts and to be as compact as possible in order to minimize transmission time and occupied memory space.

# CHAPTER V

## FPL/I: A CAPS FLOWCHART PROGRAMMING LANGUAGE

Flowchart Programming Language I (FPL/I) is a language implemented in the experimental version of CAPS. FPL/I is an experiment in providing facilities within CAPS and is an example of some of the possibilities that are available. The semantics of an FPL/I flowchart program are conveyed by its symbols and their interconnections. The semantics of a symbol are conveyed by its type and the text associated with it.

### 5.1 Symbol Sequencing

An entry point to a flowchart is any symbol which has no ARROW connections to it. The entry point name is the text associated with the entry point. A flowchart entry point is "called" when its name appears in a symbol that is being executed. An ARROW symbol goes from the entry point to the first executable symbol which is executed when the entry point name is "called". An ARROW from this symbol indicates the next symbol that is to be executed and so on. No ARROW from a symbol indicates that it is an exit point from the flowchart. At an exit point, the current invocation of this flowchart is completely executed and control is returned to the "calling" flowchart. The existence of two or more ARROWs from a nonDECISION type symbol is an error. FPL/I has no provision for the parallel execution of flowcharts.

In FPL/I there is an addressable condition register which contains a name generated by a flowchart's execution. For example, "+", "0", "-", "YES", and "NO" are names that can appear in the condition register. These are names of conditions that exist for data and

operations. The name in the condition register is used to decide how to proceed from a DECISION type symbol. If the condition name matches the name on an ARROW, then this ARROW goes to the next symbol to be executed. If there are no name matches, then an ARROW with a blank name on it indicates the next symbol. If there is no ARROW with a blank name on it, then this symbol is an exit point from the flowchart.

## 5.2 Symbol Type Semantics

As was illustrated, a DECISION symbol indicates that a control decision is to be made in the execution of a flowchart. An ARROW symbol indicates the next symbol to be executed and contains the text that makes control decisions. To complete the specification of the semantics for the symbol types, a TEXT symbol is regarded as containing comments and is not to be executed. PROCESS, CONNECTOR, and DECISION symbols contain instructions that are to be executed.

## 5.3 Symbol Text

The text associated with a symbol conveys the remainder of the semantics. The text consists of references to flowcharts, primitive instructions, and data. The text is delimited with a colon (:). The primitive instructions are realized by the flowchart executor and are analogous to machine instructions. Data are floating point numbers. Parameters for primitives and flowcharts are written in prefix form. In execution, the text is scanned from left to right. This scan associates each identifier's value with the identifier. This scan is suspended by the occurrence of a ":" and a reverse scan of the text is started. The primitives and flowcharts are executed during the reverse scan which is terminated at the end of the text. At the conclusion of

the reverse scan, the initial scan resumes at its suspension point and terminates at the end of the text. Any ":" encountered in the initial scan initiates the reverse scan.

For example, in the FPL/I's text strings

```
= A SQRT + 3.7 5.3 :          (1)

  PRINT A :                   (2)
```

=, +, and PRINT are the primitives and SQRT is a flowchart which (see Figure 1) calculates the square root of a number. After executing (1), A will have the value 3.0 and after executing (2),

```
A = 3.0
```

will appear on the output media.

## 5.4 Elements

A basic building block is the element. An element has associated with it a name, type, and value. The element's name is used for the identification of the value. The element's type designates the representation for the value. The element's value is in a format specified by the type designation. In the preceding example, the element A is of type floating point number and has the value 3.0. The element PRINT is of type primitive and has a value that is a subroutine to do the print operation. The element SQRT is of type flowchart and has a value that is a flowchart to calculate the square root of a number.

The association of a value with each element makes the element similar to a conventional memory location. The association of a type with each element is not a conventional memory technique but has been done on commercial and experimental computers. For example, the B5500 [2], the GENIE computer at RICE[13] and the TX-2 at Lincoln Laboratories MIT[3], all have type bits associated with their memory locations. The

association of a name with each element makes the elements similar to locations in a content addressable memory.

Each element is either of type floating point number, flowchart, or primitive. The value of a flowchart type element is a link to an internal flowchart representation generated from a drawn flowchart. The value of a primitive type element is a "machine instruction" - a link to a program within the interpreter which simulates the machine instruction. All elements are members of two-way (forward and reverse) linked lists in order to facilitate element storage allocation, addressing, and manipulation. An element is addressed relative to a specific list. These lists are organized horizontally. Relative to a given element in a list, it is possible to find the element to its right or left. (Internally, there are elements that provide two-way linkage between lists.)

These internal lists are used for: 1. the basic storage of elements; 2. a scratchpad area for the execution of a symbol's instruction, arithmetic calculation, and element manipulation; 3. the internal representation of a flowchart; and 4. the storage of the information necessary to execute the flowcharts. The same list format is used for all these purposes. The latter uses of these lists are related to the inner workings of the interpreter and their description will not be included in this discussion.

## 5.5 Element Storage

A two-way linked list is used as a stacked name table for element storage. This stack is used and manipulated by primitive instructions. One use of the name table occurs when an identifier's value is needed for a computation. The name table is searched, from the top, for the first occurrence of the identifier name in the name table,

in order to get its value. For example, the execution of the text string

= Z + Z 8.9 :

will cause the value of Z in the name table to be increased by 8.9.

As other examples of the name table's manipulation, consider the primitives SET, for adding an occurrence of a name table, and FREE, for deleting an occurrence of a name from the name table. The SET primitive causes one new element to be "pushed" into the top of the name table stack. The FREE primitive causes the name table to be searched, from the top, for the first occurrence of the indicated name. This first occurrence of the name is deleted from the name table. Thus in the execution of the text strings

SET Z FLOAT 3 : (1)

SET Z FLOAT 4 : (2)

SET Z FLOAT 5 : (3)

FREE Z : (4)

FREE Z : (5)

the occurrence of Z FREEed by (4) is the one SET by (3), that FREEed by (5) is the one SET by (2) and the occurrence of Z that remains in the name table is the one SET by (1).

## 5.6 Identifier Name, Identifiers, and Identifier Scope

An identifier name contains at most six alphanumeric characters and the first character is a letter. An identifier is a particular occurrence in the name table of an identifier name. An identifier is defined with the SET primitive during the execution of a flowchart program. For example, the execution of the text string

SET Z FLOAT 8.9 : (1)

creates the identifier Z as a floating point number. An identifier is active until it is deleted or until another identifier with the same name is created. In the latter case, the most recently created identifier is active and all other identifiers with the same name are passive. The execution of the text string

SET Z FLOAT 73.0 :

creates an active identifier Z and makes the Z of (1) passive. An identifier is deleted with the FREE primitive during the execution of a flowchart program. For example, the execution of the text string

FREE Z :

deletes the currently active identifier Z from the name table. The chronologically youngest passive identifier Z is now the active identifier Z.

The scope of an identifier is the time that the identifier is active. Thus an identifier's scope is defined by the creation and deletion of identifiers by executing flowchart programs.

## 5.7 Execution of a Symbol's Text

The processing list is used as a scratchpad area for the execution of a symbol's instruction. This execution takes place in the following way: the symbol's text string is scanned a character at a time from left to right until a ":" is reached. As this scan proceeds, the identifier names and the numeric literals are detected and cause the insertion of new elements into the processing list. For an identifier, the name of this new element is the identifier name, the type indicates its origin, and there is no value associated with this element type. For a numeric literal, the new element's name is blank, the type is floating point, and the value is the number. This text scan is suspended when a

":" is found and a right to left examination of the processing list is initiated. This corresponds to a right to left scan of the text string. This examination is done an element at a time and is terminated at the end of the processing list causing the text scan to be resumed.

The purpose of this element by element examination is: 1. to search the name table for a type and value designation for the element name, 2. to update the processing list element with this new information, and 3. to execute references to flowcharts and primitives. For example, in the initial scan of the text string

$$= \underline{Z}\ \underline{SQRT}\ \underline{+}\ \underline{3.7}\ \underline{5.3}\ \underline{:} \qquad (1)$$

six elements are generated and inserted into the processing list. These elements correspond to the groups of characters that are underlined in (1). This scan is suspended by the ":" and the execution of the accumulated list of elements begins.

The first two elements encountered in this examination are the floating point numbers 5.3 and 3.7 and they are left in the processing list as the examination advances to the + element. The type and value of the first occurrence of + in the name table indicate that + is a primitive. The primitive + adds the values of the two elements to the right of its element in the processing list and inserts the sum as a new element into the processing list. The name of this new element is blank and it is of floating point number type. The elements representing the two input arguments and + are deleted from the processing list and their place is taken by the new element. SQRT is the next element that is examined. The information in the name table indicates SQRT is a flowchart (see Figure 1 for SQRT's flowchart). SQRT takes the square root of the value of the element that is to the right of its element in the

processing list. A new element with a blank name, floating point number type, and value that is the computed square root is inserted into the processing list in place of the deleted elements SQRT and 9.0.

The element Z is a floating point number and its type and value are left in the processing list as the examination advances to the = element. The primitive = alters an entry in the name table. The element to the right of the = element provides the identifier name. The type and value of the second element to the right are stored in the name table with this name. The elements for =, Z, and 3.0 are deleted from the processing list. (See Figure 14 for a description of the processing list at each of the critical stages.)

This was an example of the way in which 1. a text string is analyzed, 2. elements are created from the analysis of the text string, 3. created elements are inserted into the processing list, 4. primitive and flowchart instructions are executed, and 5. parameters are manipulated in the processing list.

## 5.8 Flowchart Parameters

All of the "called" flowchart's parameters, input and output, are provided "by value" in the "call by value" sense of ALGOL[18]. A parameter's value is either a flowchart, primitive, or floating point number. A flowchart valued parameter, because it is executable, provides parameters "by name". Thus, the "calling" rather than the "called" flowchart determines that a parameter is provided "by name" or "by value". The FPL/I "by name" parameters are slightly different from the ALGOL "by name" parameters; a discussion of the two types of "by name" parameters will be presented later.

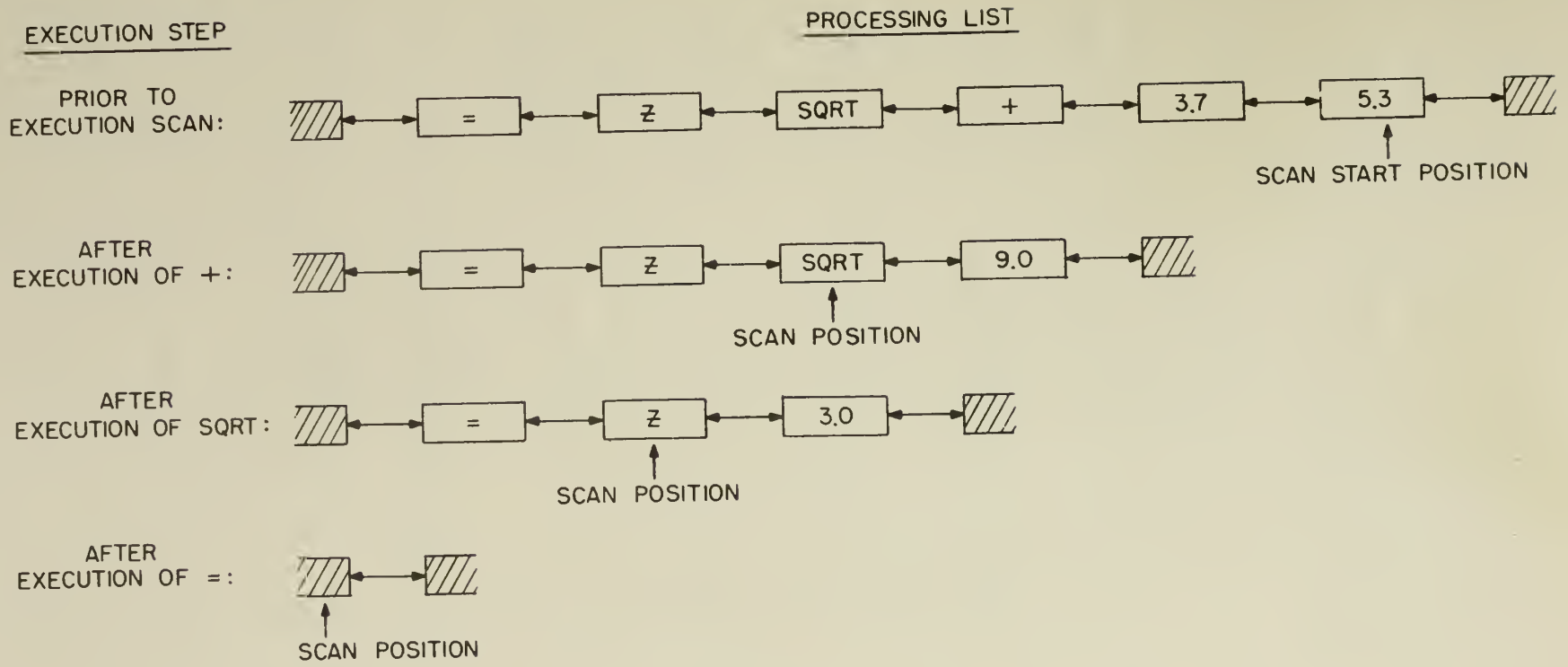



FIGURE 11
PROCESSING LIST DURING THE EXECUTION OF A TEXT STRING

Flowchart parameters are provided in the processing list and in the name table. The "calling" flowchart puts input parameters into the processing list by placing the parameter identifiers and expressions after the flowchart's identifier in the "calling" text string. For example in

= X INTG START STOP <POLY> : (1)

the parameters for the flowchart INTG (see Figure 15) are START, STOP, and POLY. START and STOP are floating point number. POLY is a flowchart (see Figure 16) and ">" is a primitive that allows POLY to be input as a flowchart value rather than being executed.

The character "<" delimits the scope of ">". The result of executing ">" leaves one input argument, the value of POLY, in the processing list. POLY calculates the value of a function for use by the flowcharts INTG and INTG1. The parameters for the primitive = are X and the single element output from INTG. START and STOP are "by value" parameters, POLY is a "by name" input parameter. Each reference in INTG to this parameter results in the execution of POLY relative to the active identifiers in INTG.

The "]" primitive creates identifiers and assigns to them types and values that are found in the processing list. The "[" delimits the scope of "]". The "]" primitive is used by the "called" flowchart to accept input parameter values and assign identifiers to them. For example in (1), the flowchart INTG is invoked with the parameters START, STOP, and POLY. In flowchart INTG at (2), "]" takes the input parameter values from the processing list and assigns them to the identifiers A, B, and F, respectively. The RETURN primitive (3) deletes the identifiers created by "]". The flowcharts INTG and INTG1 (see Figure 15) evaluate

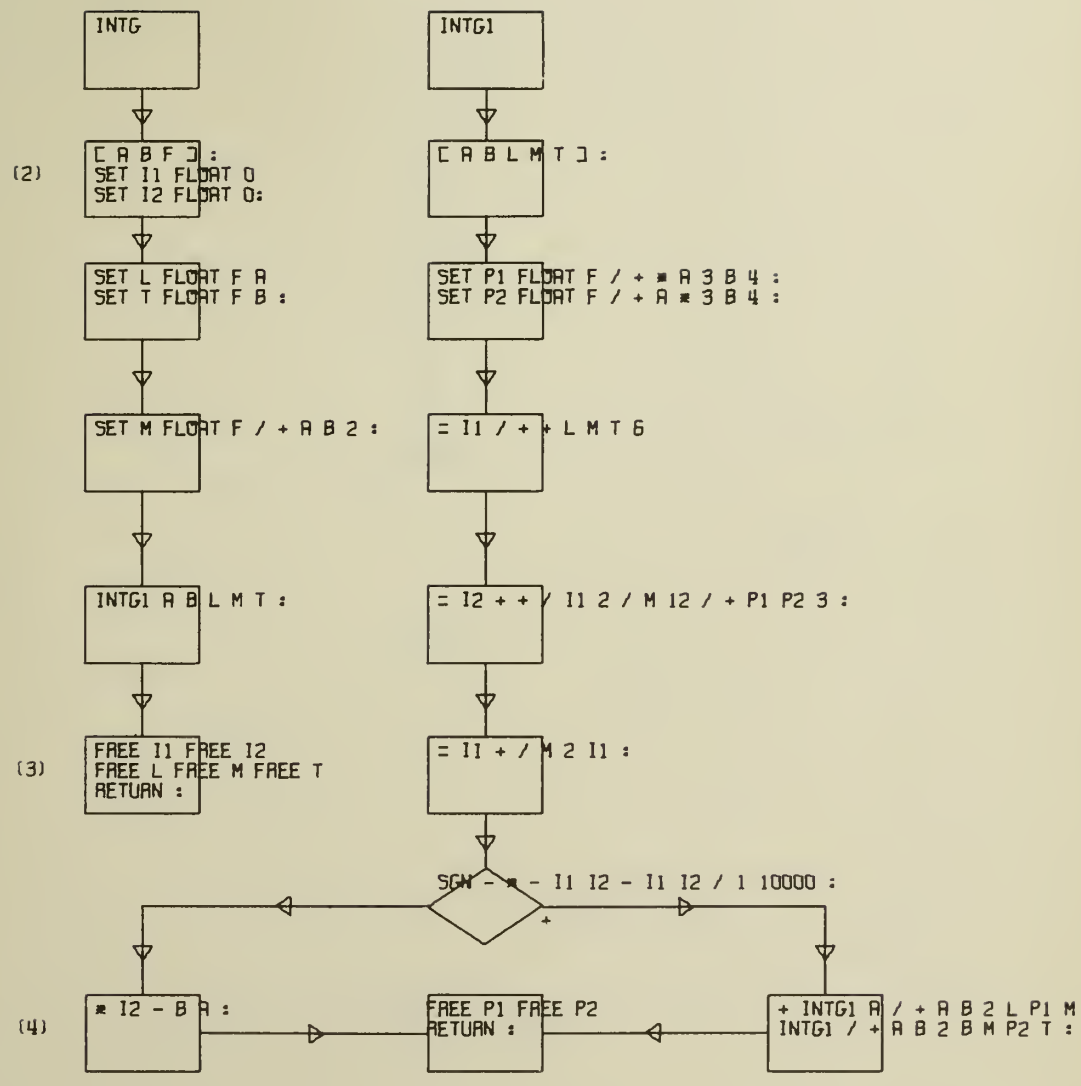
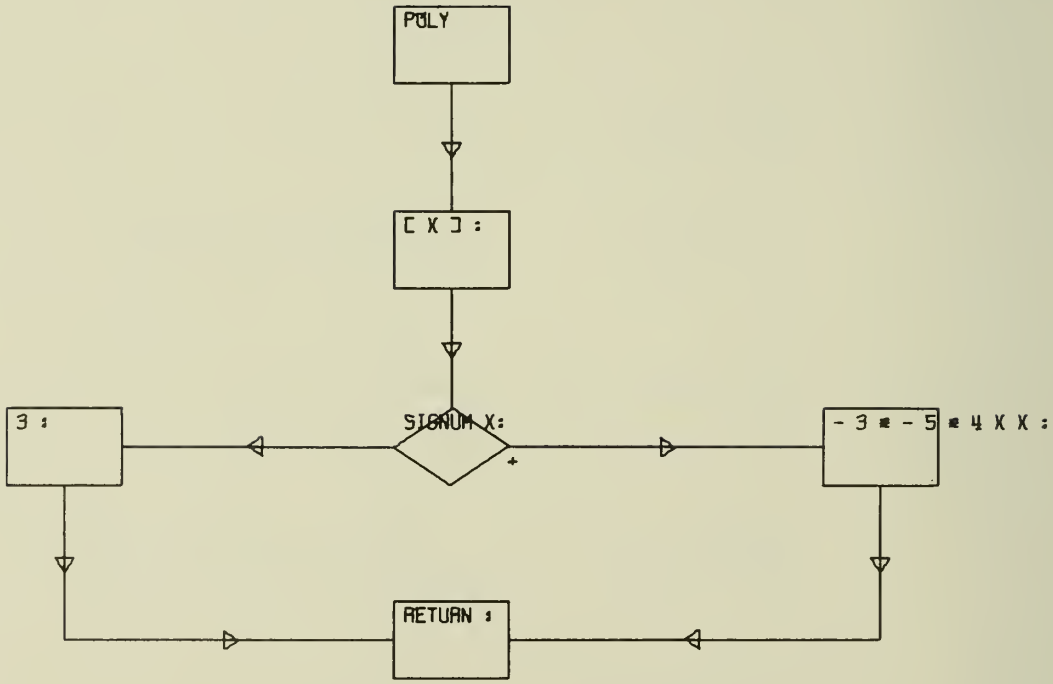



Figure 15. INTG and INTG1 Flowcharts




```
POLY (X) = 4*X**2-5*X+3 IF X > 0
         = 3 IF X < 0A = 0
```

Figure 16. POLY Flowchart

an integral using Simpson's rule with variable step size. INTG1 computes the integral by means of one Simpson rule step and two Simpson rule steps over half the interval. If these are close then the second is taken as the answer, otherwise INTG1 is used recursively to evaluate the integral over each half interval separately.

The "called" flowchart puts output parameter values into the processing list by inserting more elements into the processing list than are taken out. The elements are inserted into the processing list from a symbol's text. Elements are inserted and deleted as a result of executing flowcharts and primitives. For example, in INTG1 the "by value" output parameter is the result of executing the text string (4). The value of (4) is returned to the "calling" flowchart and is used there as input to a flowchart or primitive.

The "by name" parameters allow a flowchart to accept identifier names as input, use their values, and associate outputs with them. For example in Figure 17, the "calling" flowchart invokes flowchart Y with the input parameter S. The output from S is the element B and it is inserted into the processing list. In the "called" flowchart Y, A's value is the same as S's. In (5), the reference to A causes S to be executed which outputs the element B and then the primitive + adds the values of B and 3. In (6), the reference to A outputs the element B and then the primitive = gives B the value 4.

Parameters are provided in the name table with the use of a common identifier. A common identifier is the device of having the "called" and the "calling" flowcharts agree that they both will use the same identifier for a specific value. For example, if in the "calling" and "called" flowcharts X and Y the identifier Z is active, then X stores

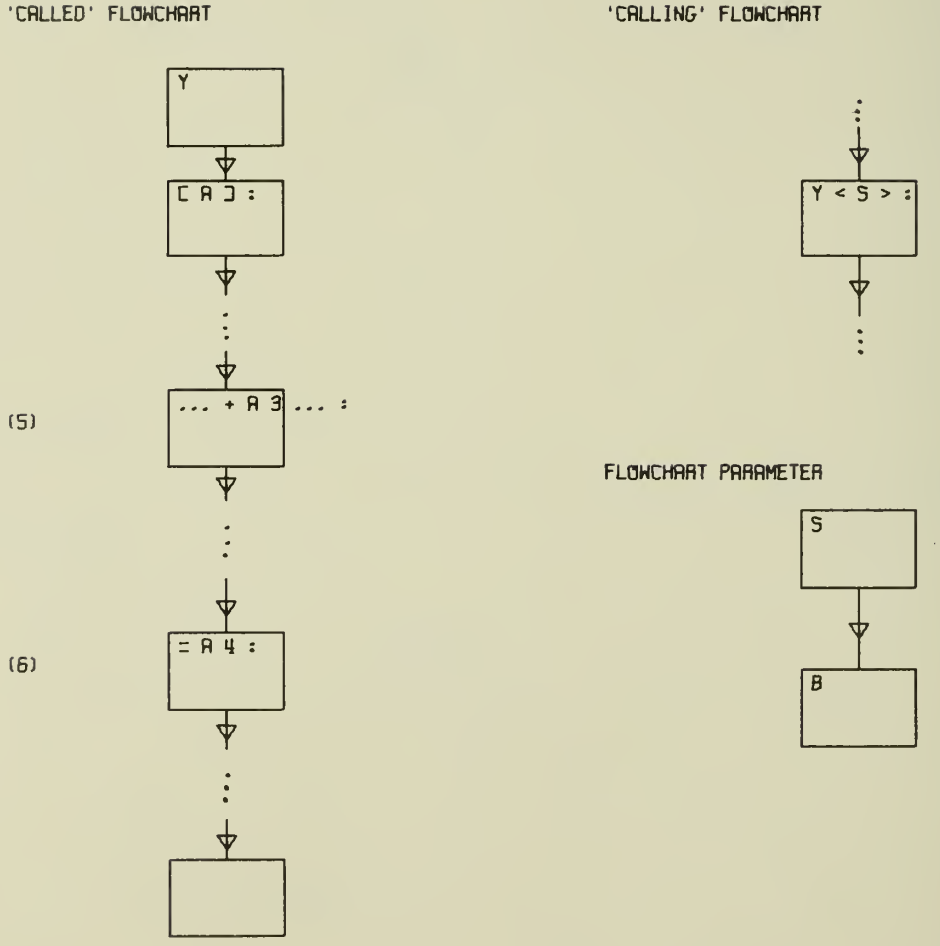



Figure 17. "By Name" Parameterization

a value in Z, calls Y, and Y uses the value as an input parameter. Output is done in a similar manner. This is slightly different from the EXTERNAL attribute of PL/I[15] and the COMMON declaration in FORTRAN[14].

## 5.9 Condition Register Primitives

As was previously discussed, the condition register contains a name that is used to make a control decision. A set of primitives are used to store names in this register. For example, in

```
SIGNUM Z :
```

SIGNUM is a primitive that stores "+", "-", or "0" in the register if Z is a positive, negative, or zero floating point number. The primitive SGN stores the same names but the argument is an expression. For example in

```
SGN - * 3 4 * 5 6 :
```

SGN stores a "-" in the register.

The primitive OVFLOW stores "YES" or "NO" if the last floating point operation did or did not generate an overflow.

## 5.10 Input/Output

The Input/Output primitives are READ, PRINT, and READER. One floating point number is input by READ and output by PRINT. For example in

```
READ A :
```

READ prints

```
A = ?
```

on the output media and then reads a line from the keyboard. The line is executed, in the same way that symbol text is executed, and leaves a

result in the processing list. Then, A is given the value of this result. If

```
- * 6 6 * 7 7 :
```

is input, then A will receive the value -13. The primitive READER reads a line from the keyboard and executes it. This primitive allows a user to print out arbitrary values, alter identifier values and alter the processing list.

5.11 Debugging

Flowchart programs are debugged with tools provided by FPL/I and CAPS. Some of these are similar to existing tools for conventional on-line debugging. Others of these tools owe their existence to the use of flowchart programming languages.

The user is able to initiate, suspend, resume or terminate the execution of a flowchart program. In the event of the generation of an error condition, the symbol on the flowchart program causing the error is displayed. The user is able to "trace" the actual or slowed down execution of a displayed flowchart by watching the blinking of the currently executing symbol. In addition, the user is able to "single step" the executing program. While the execution of a flowchart is suspended, the user can: 1. display and/or alter the value associated with any element; 2. display the ordered sequence of flowchart names that are in the process of being executed; 3. terminate the execution of these flowcharts; and 4. install "break points" within a symbol or between symbols thereby allowing the flowchart to execute in a non-traced manner until a break point is reached.

Other FPL/I debugging features include: 1. each time or each n times that a particular identifier is referenced in any way or is

referenced in order to overstore the element value, the execution of the program is temporarily suspended, the value of the identifier is displayed, and the CAPS user given the opportunity to use any of the available debugging features; 2. the ability to manipulate input/output parameters easily in order that a particular flowchart may be run and debugged without requiring "calling" or "called" flowcharts; and 3. the ability to indicate, by blinking symbol display, all the symbols whose text contains an occurrence of a particular text string.

Another debugging feature that has received consideration is the ability to distinguish a particular set of identifiers prior to or during the execution of a program, completely or partially execute the program, and then discover the influence of any of the distinguished identifiers upon any arbitrary identifier. For example, in a flowchart program to calculate the value of a complicated function F of three arguments X, Y, Z, the CAPS user could distinguish the element Z, execute the flowchart program for F, and then discover the "dependence" of F (X,Y,Z) upon Z. This dependence could take the form of a numerical approximation to $\frac{\partial F\ (X,\ Y,\ Z)}{\partial Z}$. This type of debugging aid has not been implemented in CAPS for FPL/I.

## 5.12 Pointer and Pointer Manipulation

An addressable pointer is provided for the manipulation of the internal lists. This pointer "points to" an element in a list. The pointer is moved right or left one element with the primitives PRIGHT or PLEFT. If the "pointed to" element is an internally defined element with two-way vertical links to other elements, then the pointer can be moved up or down one element with the primitives PUP or PDOWN. The pointer is given a value with the primitive PSETC. The element

"pointed to" by the pointer is printed on the output media with the PSHOW and PSHOWX primitives. PSHOW prints out the name and value of the element. PSHOWX prints out the element's contents in octal format.

This pointer is used, for example, in printing out the contents of an internal list. The text string

LIST NAMET <PSHOW> :

invokes the flowchart LIST (see Figure 18) to print out the name table. (See Appendix C for a detailed description of FPL/I primitives.)

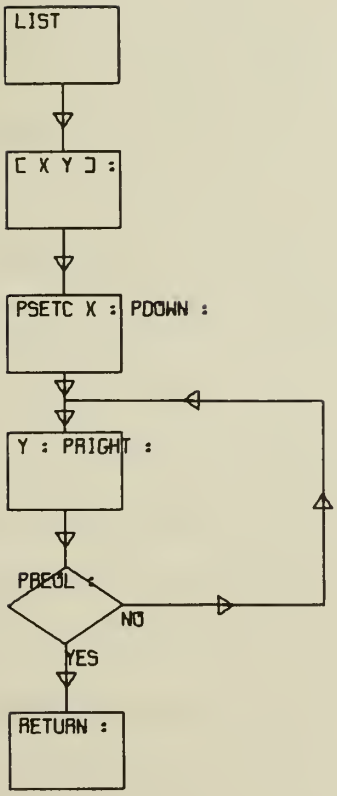



Figure 18. LIST Flowchart

# CHAPTER VI

# OBSERVATIONS AND NOTES ON FPL/I

The following observations are based upon the experimental use of FPL/I within CAPS. Included in these observations are notes on suggested additions and refinements for FPL/I and CAPS.

## 6.1 Inter Computer Communication

The performance of the CAPS computer system is affected by the speed of the communication link between the display device and the supporting computer. The speed of the link in the experimental system is 100 characters per second. This means that a screen full of text (2048 characters) requires twenty seconds for transmission between the two computers. The time required to transmit the representation of a flowchart of thirty symbols is approximately twenty seconds. The time required to transmit a flowchart of 120 symbols is approximately 80 seconds. All transmission times depend upon the amount of text in the flowchart's symbols and upon the response time of the supporting computer.

The transmission of an entire flowchart is a time consuming thing. The number of times that an entire flowchart representation is transmitted may be reduced by piece wise transmission of the representation or by providing bulk memory for the display device. With piece wise transmission, the supporting computer would be notified of each flowchart change as it is made. This would allow both computers to maintain identical representations. Also, the transmission time for the change information is masked by the fact that the programmer is doing more flowchart construction and editing while the transmission takes place.

However, piece wise transmission does not solve the problem when the supporting computer is required to transmit an entire flowchart representation to the display device. For this, the programmer must wait for the transmission of the entire flowchart. This fact affects flowchart editing, the ability of the system to trace more than one flowchart at a time, and the ability of the system to allow the definition of multilevel flowcharts. For flowchart editing, the flowchart must be transmitted from the supporting computer to the display device before it can be altered.

For CAPS to trace the execution of more than one flowchart at a time, each flowchart must have its representation transmitted to the display device before its execution can be traced. This transmission must take place each time that the flowchart is entered from another flowchart. Thus the execution tracing of many flowcharts would be a time consuming task. An analogous problem exists in the question of allowing the creation of multilevel flowcharts (flowcharts within flowcharts). A multilevel flowchart is one in which a symbol may be EXPANDed into a flowchart and a flowchart CONTRACTed into the symbol of another flowchart. For this facility, an entire flowchart representation must be transmitted from the supporting computer to the display device for each EXPAND or CONTRACT because the flowcharts can't all be kept in the display device's memory. This transmission is a very time consuming task. In the proposed display device configuration, the transmission time element rules out the execution tracing of many flowcharts and the facility for creating multilevel flowcharts.

For the sake of argument, let us suppose that the display device had bulk memory in the form of disk or drum. Then, duplicate

flowchart representations would be kept, one in each computer. This would allow the flowchart's name to be transmitted between the two computers, rather than its representation, which would speed up flowchart tracing and editing operations. It would also allow the construction of multilevel flowcharts.

An entire flowchart would be transmitted only when one computer had a flowchart representation that the other computer did not have. This is the case when either computer is used independently of the other. Thus with the addition of bulk storage to the display device, the importance of the communication link's speed is reduced.

In the experimental system, the display device has magnetic tape storage (DECtape). Flowcharts are constructed and stored on the DECtape if the supporting computer is not available. At a later time, these flowcharts can be transmitted to the supporting computer. The communication link's speed is a limitation only for this transmission. Multilevel flowcharts (EXPAND and CONTRACT operations) are not feasible with magnetic tape because of the tape's sequential access and relatively slow word transfer rate. (The DECtape played a vital role in the development of the system in that display device programs and flowcharts were stored at the display device rather than in the supporting computer.)

One of the constraints on CAPS is that the display device hardware not be too expensive. The cost of the experimental system is:

| | |
|---|---|
| Basic Display Device | $ 55,500 |
| Additional 12,288 Words of Core Memory | 24,650 |
| Character Generator | 6,000 |
| DECtape Control Unit | 10,100 |
| (2) DECtape Transports | 4,700 |
| Remote Transmission Hook-up | 1,000 |
| Spares | 7,225 |
| Total | $109,175 |

In this, the cost of the DECtape is 27% of the basic price and 18% of the extended memory price. The projected costs for display devices in third generation hardware start at $10,000[10]; the cost of bulk memory will not soon go below $6,000. Eventually, the cost of bulk memory may be double the cost of the display device. For this reason, there is no requirement that a CAPS display device have bulk memory.

## 6.2 Flowchart Construction and Editing

The basic set of drawing operations (MOVE, DELT, and TEXT) were provided in the initial version of the flowchart editing system. The allowed size of the flowchart (10" x 10"), the ability to move only one non-ARROW symbol at a time, and the absence of vertical or horizontal alignment operations were increasingly irritating restrictions as experience was gained in using the system.

The limited 10" x 10" area for flowchart construction allowed not more than approximately 120 PROCESS and DECISION symbols to be closely packed onto a flowchart. The space needed for indicating the topology of the flowchart was more of a problem than was the allowed number of symbols. For example, in a reasonably complicated flowchart

program of about 50 symbols, the ARROWs appear to fill up the flowchart display. This is because the ARROWs may vary in length and must connect diverse parts of the flowchart. In FPL/I, a flowchart is defined as the flowchart representation that is contained in the display device's memory. In the initial system, this meant that all flowcharts had to be broken up into flowchart "subroutines" that were no larger than 10" x 10".

All of these factors led to the decision to provide for the drawing of larger flowcharts. An allowed 40" x 40" size was decided upon because the required symbol coordinates are 12 bits and that is the width of the display device's memory in the experimental system. Selected portions (40" x 40", 20" x 20", or 10" x 10") of the flowchart are viewed through the CRT window. The selected part of the flowchart is scaled down to fit the 10" x 10" CRT by the operations SC X1, SC X2, and SC X4. The window is moved relative to the large flowchart by the operations LEFT, RIGHT, UP, and DOWN. Also a "joy stick" is available for moving the window. For the purpose of moving the window, the large flowchart is projected onto a torrus. Thus, there are no edges to the large flowchart and the window can be moved, indefinitely, in any direction. The flowchart may be edited in any scale and with the window in any position.

The three sizes of viewable portions of the flowchart correspond to three viewing scales. The decision to provide only three fixed viewing scales rather than a "continuous" range of scales was reached because of the lack of multiply/divide hardware on the experimental display device. Also, the three fixed scales simplify the implementation with resulting savings in memory space and response time. The experience

gained in using this system indicates that for this size flowchart, the three viewing scales are adequate. However, any increase in the size of the allowed flowchart would require an increase in the number of viewing scales.

In the initial version of the system, the only way that two symbols could be aligned on a vertical or horizontal line was to move one of the symbols until they both were aligned. These alignments have no effect on the flowchart's execution, but give the flowchart CRT display and incremental plotter hardcopy versions neater appearances. The VERT and HORZ alignment operations were added to later versions of the system as rapid methods of aligning symbols. These operations affect the current symbols' coordinates but do not constrain the symbols to remain horizontal or vertical in future moves.

In the initial version, only one non-ARROW symbol could be moved at any one time. This made the rearranging of entire sections of a flowchart a tedious operation because each section had to be moved one symbol at a time. For this reason, the "group" operation was put into later versions of the system. The "group" operation allows the user to define one set of symbols as a "group".

If a member of the "group" is moved, then the entire "group" is moved. This operation allows sections of flowcharts to be moved in the same way that a symbol is moved. If a member of the "group" is deleted from the flowchart, then the entire group is deleted from the flowchart. If a member of the "group" is duplicated with the ADD operation, then the entire "group" is duplicated. In this sense, the "group" operation is a drawing tool.

In the initial version of CAPS the flowchart editing system fit into 4096 memory words in the display device. The later versions of the system occupy all 16,384 memory words.

## 6.3 Identifier Scope

The scope of an FPL/I identifier is defined by a flowchart's execution sequence. This definition is different from that used in conventional languages. For example in Algol, an identifier's scope is determined, prior to "execution time", from the relative position of its declaration. In the ALGOL procedure AA (see Figure 19), procedures AA and CC are the scope of the identifier Z declared in AA. Procedure BB is the scope of the identifier Z, declared in BB. The interesting thing about this is that procedure CC is the scope of identifier Z, declared in AA, even when CC is called from BB. This is not the case in an analogous situation in FPL/I. In the FPL/I flowchart AA (see Figure 20), the identifier Z created in AA is active in CC when CC is invoked from AA. The identifier Z, created in BB, is active when CC is invoked from BB. Identifier Z's scope is not the same as in the ALGOL procedure AA. This ALGOL identifier scope is not constructable in FPL/I. This example indicates a fundamental difference between the two methods of defining an identifier's scope.

The FPL/I identifier creation method allows a flowchart to specify the allocation of storage. This is desirable when large areas of storage are required for scratch purposes and are not needed as permanent storage. Also, each FPL/I identifier is a "stack" that is "stacked" with the SET primitive and "popped" with the FREE primitive. This "stack" attribute of an identifier allows the same identifier name to be

```
PROCEDURE AA ;
     REAL Z ;
     PROCEDURE BB ;
          REAL Z ;
          .
          .
          .
          CC ;
          .
          .
          .
     END BB ;

     PROCEDURE CC ;
          .
          .
          .
          Z = ... ;
          .
          .
          .
     END CC ;
     .
     .
     .
     BB ;
     .
     .
     .
     CC ;
     .
     .
     .
END AA ;
```

Figure 19. ALGOL Procedures AA,BB, and CC

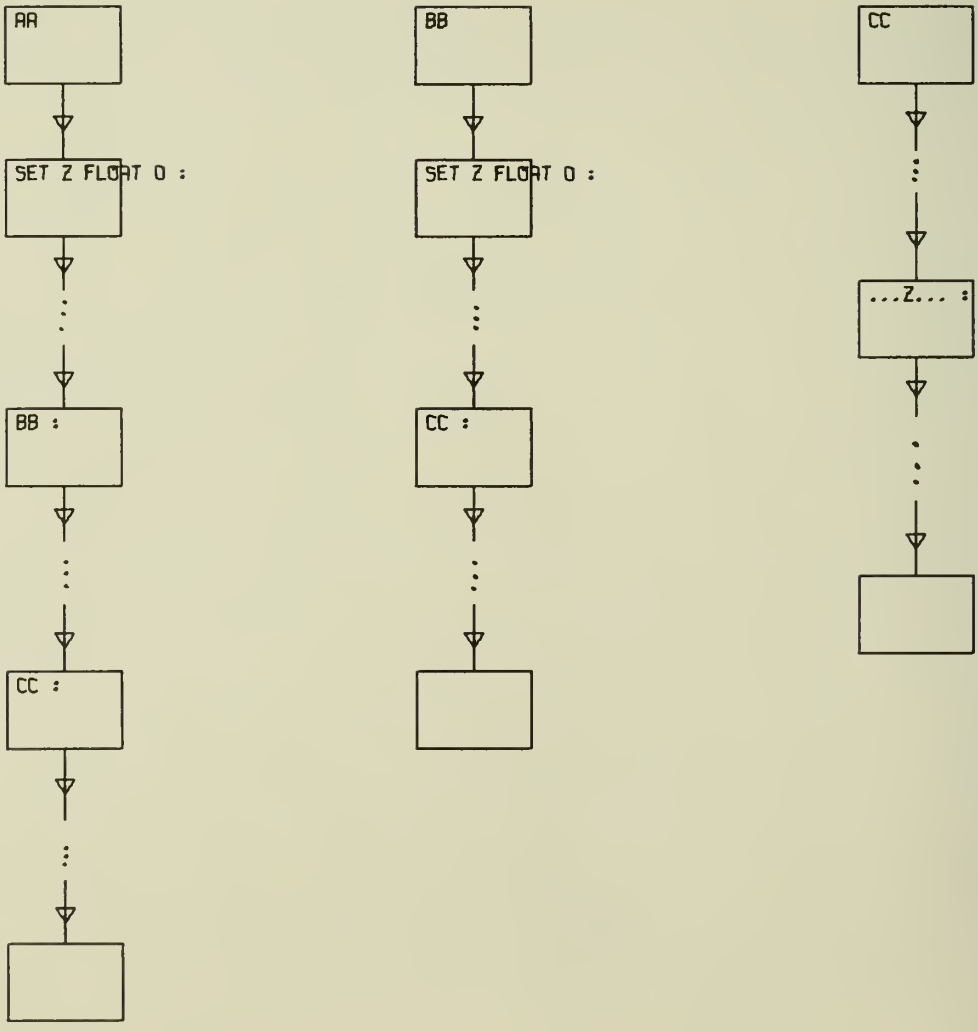



Figure 20. FPL/I Flowcharts AA,BB, and CC

used for calls upon flowcharts while the values of passive identifiers with the same identifier name are retained undisturbed in the name table.

The FPL/I scope definition makes it impossible to provide flowchart parameters by the use of common identifiers in certain situations. For example in the FPL/I flowcharts D,E, and F (see Figure 21), the identifier X is created both in D and in E. In the execution of F, invoked from E which in turn was invoked from D, the active identifier X is the identifier created in E. The identifier X, created in D, is passive in F which means that D and F cannot use the common identifier X as a flowchart parameter.

A possible modification to FPL/I's method for identifier scope definition is to define an identifier's scope such that in flowchart AA of Figure 22, the scope of Z is equivalent to the scope of Z in the ALGOL procedure AA of Figure 19. For this equivalence, the ARROW symbols with the text string "DEFINE" on them go to a group of symbols that is the definition of a contained flowchart. The flowchart's entry point is the symbol that the ARROW comes from. The entry point name is the text in the entry point. The flowchart in which this definition appears is the containing flowchart. An identifier is local to a flowchart if it is created in that flowchart. An identifier is non-local to a flowchart if it is created in a containing flowchart. The scope of an identifier is the flowchart that it is created in and any contained flowcharts that do not declare an identifier with the same name. This scope requires that each flowchart have its own name table. Each individual flowchart name table is linked to its containing flowchart's name table (see Figure 23 for the name tables of the flowcharts in Figure 22). A

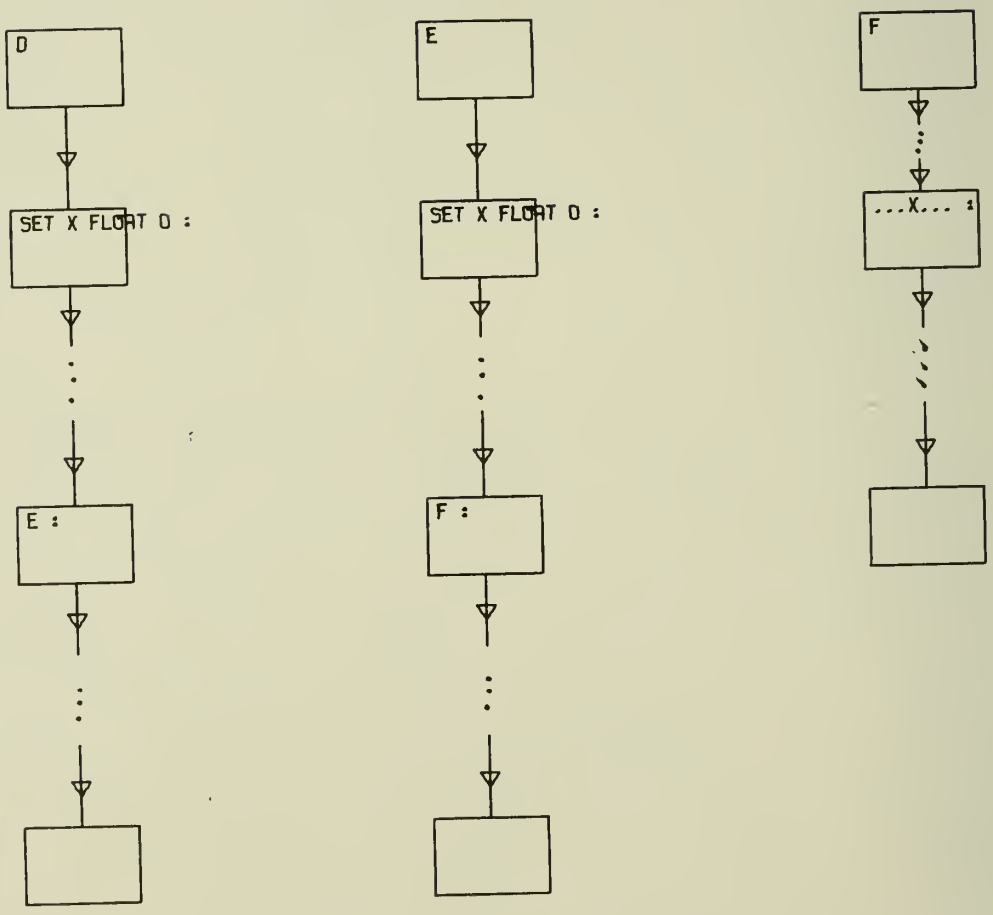



Figure 21. FPL/I Flowcharts D,E, and F

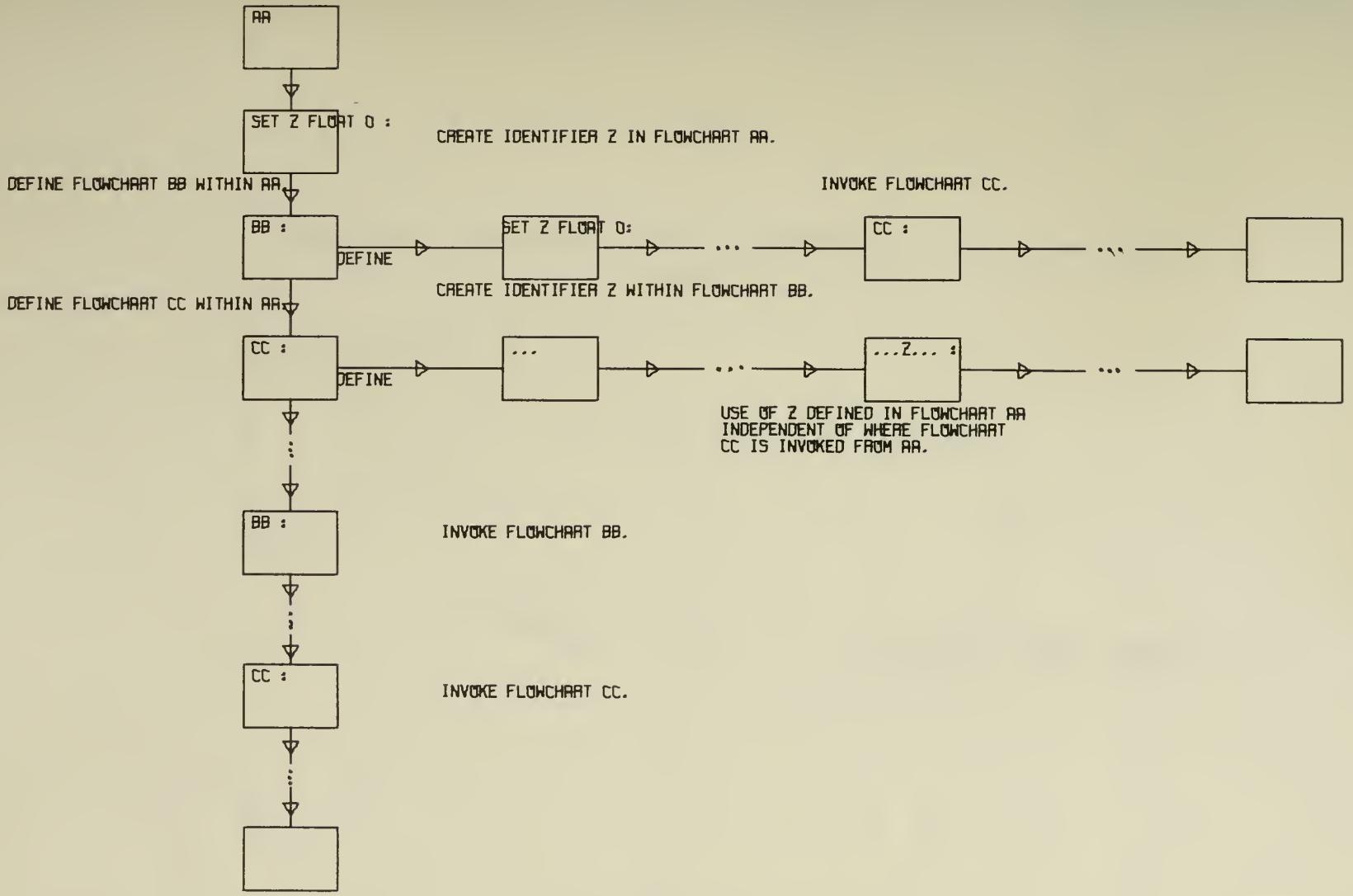



Figure 22. Flowcharts AA,BB, and CC in a Modified FPL/I

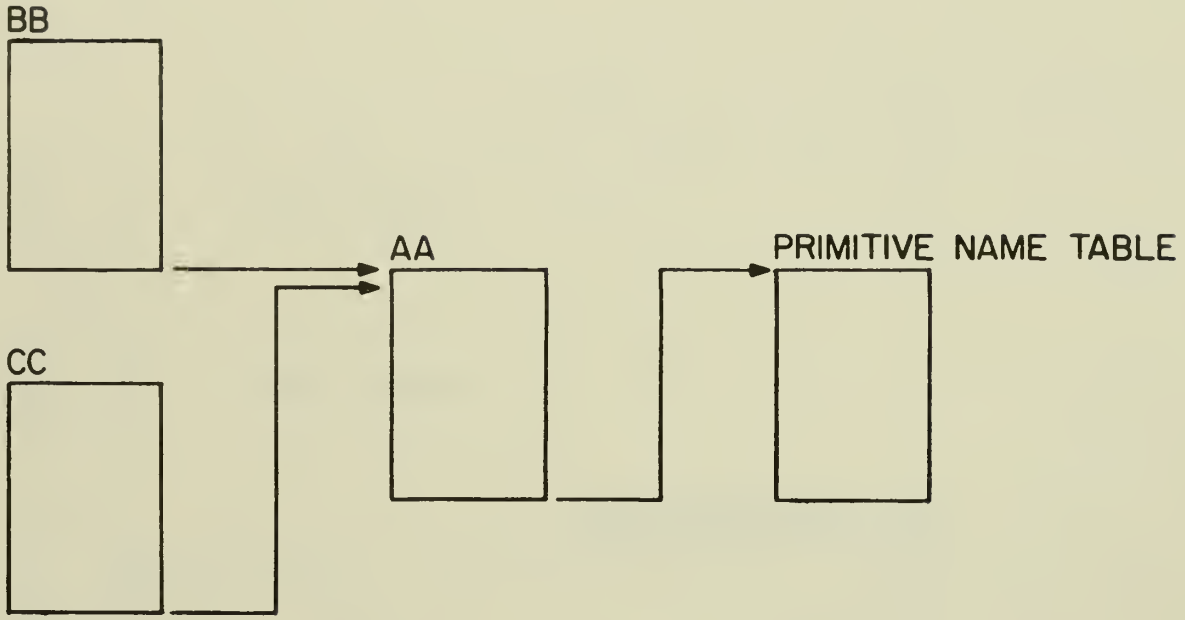



FIGURE 23
INDIVIDUAL FLOWCHART NAME TABLES AND THEIR LINKAGES

flowchart's local identifiers are created by primitives and are "stackable" in the flowchart's name table. On one flowchart, these identifiers are equivalent to FPL/I's identifiers. The active identifier is the first instance of the identifier in the name table. If an identifier is not found locally, then a search is made of the linked name tables. This allows the identifier scope of ALGOL by means of the execution defined scope similar to that of FPL/I.

Another possibility for the definition of identifier scope in a flowchart language is an adaptation of the EXTERNAL feature of PL/I. The idea is to make an identifier available to any flowchart independent of its containing flowchart. A flowchart is included in the scope of an EXTERNAL identifier when the flowchart creates the identifier with an EXTERNAL declaration.

Another possibility is the use of the graphical indication of an identifier's scope. For this purpose, a flowchart is drawn that indicates the scope of each identifier. This flowchart is in addition to the set of flowchart programs that are loaded into the supporting computer for execution. For example, Figure 24 is a "flowchart" which indicates that: 1. flowcharts X, Y, and Z are the scope of the identifier A; 2. flowcharts X and Z and flowchart Y are the scopes of the two identifiers B, respectively; 3. flowcharts X and Y and flowchart Z are the scopes of the two identifiers C, respectively; and 4. flowcharts X, Y, and Z are the scopes of the three identifiers D, respectively. This flowchart in its basic form could be provided by CAPS and displayed on the display device's CRT. Then the programmer could connect the various identifiers, thus indicating their scope.

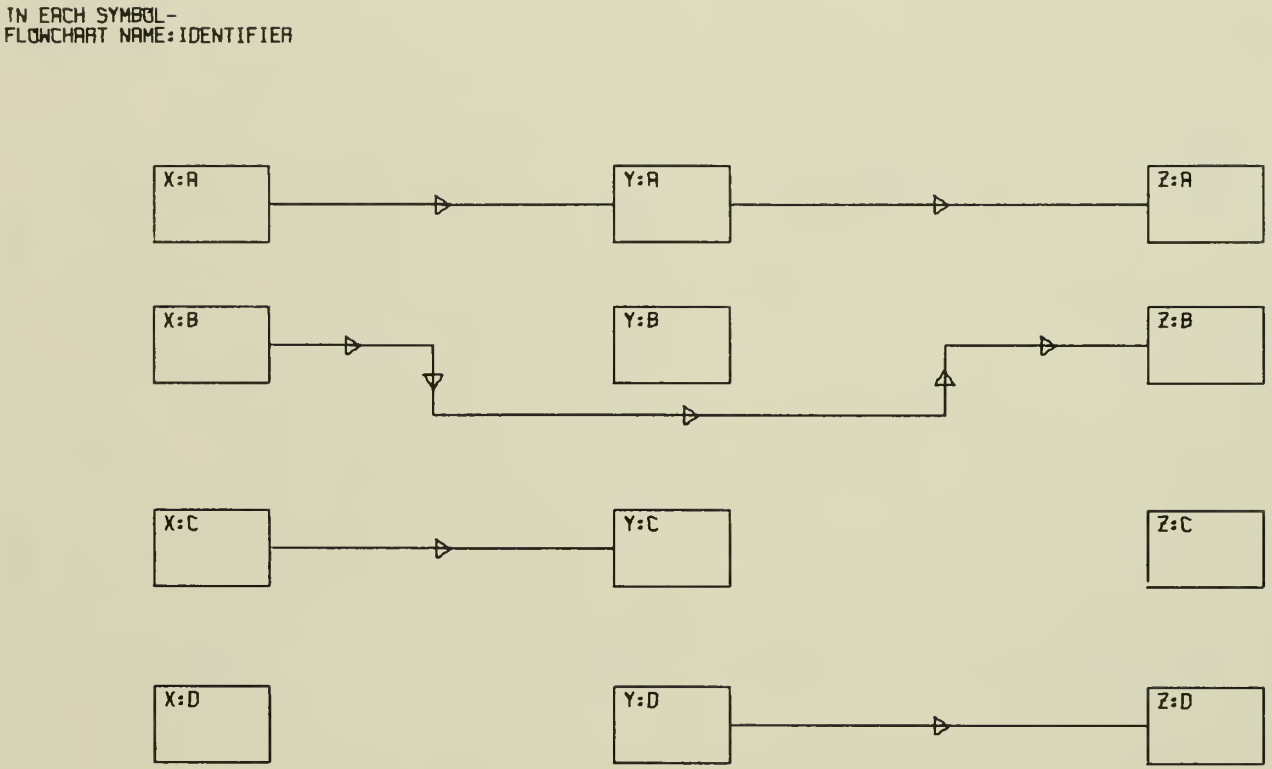



Figure 24. Graphical Identifier Scope Declaration

All three of these methods of defining identifier scope can be incorporated into the one scheme which is a reasonable approach for future flowchart programming languages.

## 6.4 Primitives and the Prefix Notation

Each primitive was designed to be as simple as possible. For example, the + and * primitives each require two inputs, have one output, and are independent of each other. This is in contrast to the conventional infix operations + and * where these operators are related by their arithmetic hierarchy. This hierarchy is demonstrated by the fact that the expression

A + B * C

is defined to be equivalent to the expression

A + ( B * C )

rather than equivalent to the expression

( A + B ) * C .

The prefix notation reflects the independence of one primitive from another. Prefix notation works well for the primitives like PRINT, FREE, or SET; but it is awkward for use in arithmeitc expressions. For example,

+ * + 3 4 5 6

( 3 + 4 ) * 5 + 6

3 4 + 5 * 6 +

are equivalent expressions in prefix, infix, and postfix notations. The most familiar is infix notation and the more awkward are prefix and postfix notations. The comparison of the awkwardness of prefix and postfix is an open question.

One way to overcome the awkwardness of the prefix primitive notation is to accept an arithmetic expression in infix notation and then have a primitive or a flowchart convert the expression into prefix notation.

## 6.5 The Execution Scan and the Delimiter

A left to right identifier recognition scan of the text string is made prior to its execution. Elements corresponding to each identifier or literal are inserted into the processing list. When a colon (:) is reached, this scan is suspended and an execution scan of the processing list is initiated. The execution scan corresponds to a right to left text string scan for primitives and flowcharts.

The right to left execution scan does not require that the primitives be recursive whereas a left to right scan would require recursive primitives. In a right to left scan of

$$= Z * + * + 3\ 4\ 5\ 6\ 7\ : \qquad (1)$$

for example, the arguments for + are 3 and 4; the arguments for * are the sum of 3 and 4 (represented by (3 + 4)) and 5; the arguments for + are ((3 + 4) * 5) and 6; and the arguments for * are (((3 + 4) * 5) + 6) and 7. In a left to right scan of (1), the arguments for * are + * + 3 4 5 6 and 7. The substring + * + 3 4 5 6 must be executed to provide one of these arguments. Thus the primitives must be recursive.

The right to left execution scan was chosen in the interest of "speed and efficiency". The colon delimiter provides a method of overcoming the awkwardness of the prefix notation and the right to left execution scan. If we assume that the execution scan is initiated when the identifier recognition scan encounters the end of the text string,

then the operation that is to be done first must appear last in the text string and the operation that is to be done last must appear first. For example, in the text string

PRINT A PRINT B PRINT C PRINT D

the first value printed out is for the identifier D and the last value printed out is for the identifier A. With the use of the colon in the text string

PRINT D: PRINT C: PRINT B: PRINT A:

the first value printed out is for the identifier D and the last value printed out is for the identifier A. The colon defines short text strings within longer text strings.

The colon delimiter is not needed in a left to right execution scan of the text string.

Postfix notation with a left to right execution scan is equivalent to prefix notation and a right to left execution scan.

## 6.6 Element Type

Each element's type defines a program's execution and is used to detect errors in that execution. For example, in a language with element types floating point number, fixed point number, character string, and array, the + primitive in the text string

+ A B

uses the types of A and B to define the addition operation. If A and B are of the same type, then the appropriate addition operation is performed. The operations are floating point addition, fixed point addition, character string concatenation, or pairwise addition of each array element depending on the type of A and B. If A and B are of different

types, then either an addition is defined between the two different types or one input argument must be converted into the other argument's type. In the preceding example, the addition of a floating point number A to an array B is defined as the addition of A to each array element in B. The addition of a floating point number A to a character string B is defined as the conversion of A to a character string and then, the concatenation of the two strings.

An element's type is used to detect program errors. For example, in conventional computers, there is no distinction between memory locations that contain data and those that contain instructions. If an errant program transfers control to a location containing data rather than instructions, the computer executes the data as instructions. The error of transferring to the data is obscured by the errors generated while executing the data. In an FPL/I processor, no floating point number is going to be executed as an instruction (primitive) because their elements have different types.

## 6.7 Element Type as a Debugging Tool

One type of "bug" occurring in conventional programs is the overstoring of memory locations with the wrong data. "Bugs" of this type are difficult to find because the overstoring operations occur and are not detected until later in the program's execution. The overstoring is discovered when error conditions arise because the wrong data is used. The overstoring error is obscured by its consequences.

In FPL/I, an identifier's value is given by an element. The element's value is in a format that is determined by its type.

If the element's type is changed in giving an identifier a new

value, then CAPS notifies the programmer of this fact and he decides whether or not an error condition exists. This feature detects one class of overstoring errors and is done by the "hardware" of the store instruction. However, it does not detect overstoring errors where the element's type is not changed. These errors are detected by replacing an identifier's element with a flowchart element.

This flowchart element is provided by the programmer and is a program to monitor the activity of the identifier. For example, flowchart B in Figure 25 informs the programmer of the changes in value of the identifier "B". The current and last values for B are maintained in the elements BB and BBB. (The names BB and BBB are arbitrary choices.) The value change in "B" is noted after the change is made but before the new value is used. This does not allow the programmer to pinpoint exactly where "B" is overstored but it does indicate a region of the program wherein "B" is overstored. If the flowchart B were able to examine the processing list and determine if a store operation was to take place on BB, then B could demonstrate exactly where each store operation on B takes place. Additional pointer primitives would have to be included in FPL/I to allow this type of operation.

Flowchart B in Figure 26 prints out the value of "B" each time that it is referenced and allows the programmer to see exactly where "B" is used.

The element type designation allows the arbitrary construction of these debugging aids which would otherwise be very difficult to construct.

An important note about this type of debugging is that the debugging diagnostics are external to the program being debugged. That

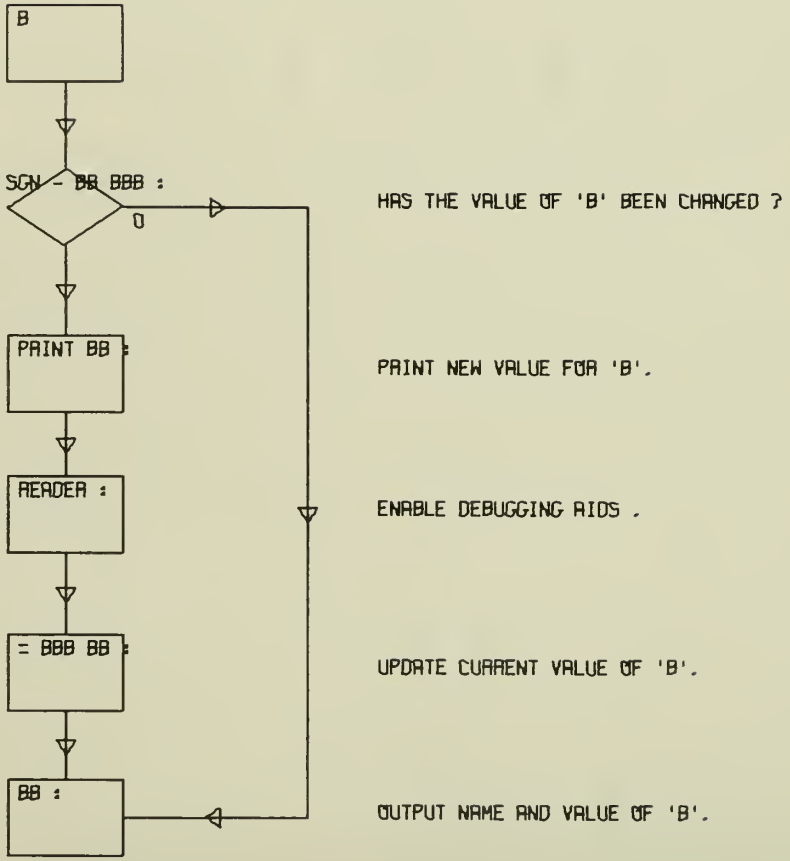



Figure 25. Flowchart for Monitoring the Changes in the Value of the Identifier "B"

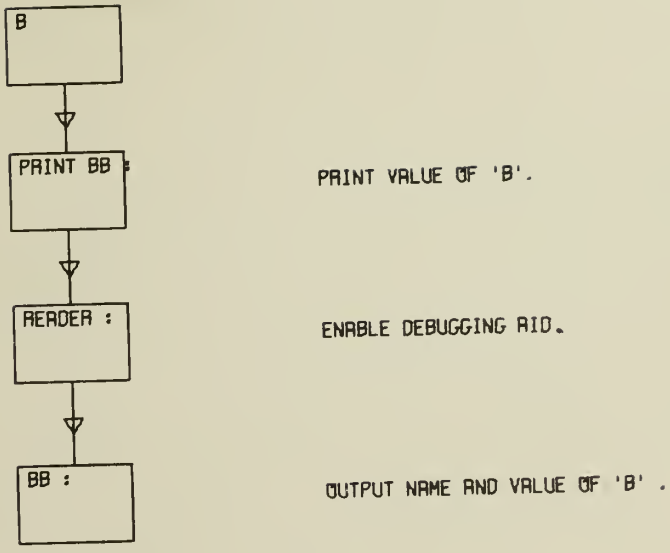



Figure 26. Flowchart for Monitoring All References to "B"

is, no changes are made to a program to include or exclude debugging diagnostics. This avoids the difficulty of inducing errors into a program in the process of removing debugging diagnostics.

Also, this scheme provides help in the case when one individual must debug a program written by someone else. The individual doing the debugging may know which identifier he is interested in but he may not be able to decide where to include debugging diagnostics in the program.

## 6.8 Flowchart Debugging

Our experience with the debugging of FPL/I flowcharts indicates that the most useful of the debugging features are the "single stepping" and "tracing" of the execution of a flowchart program, the "break point", and the ability to execute a flowchart program independent of the "calling" flowcharts. The execution of a flowchart program independent of its "calling" flowchart allows the programmer to debug a large system of flowcharts in a "bottom up" fashion. That is, the programmer debugs the most basic level of flowcharts, then when these are established as "bug free" he debugs the next level of flowcharts and so on, building up a hierarchy of debugged programs.

## 6.9 Parallel Execution of Flowcharts

For FPL/I, provision was not made for executing more than one flowchart segment at the same time. If actual or simulated multiple processors were available, then FPL/I must provide a method of specifying flowchart segments that are to be executed in parallel. Once flowchart segments are executing in parallel, provision must be made for terminating a flowchart segment's execution or allowing one segment to wait on the completion of other segments. One notation for specifying

this information is based upon the following definition: a flowchart segment is defined as a portion of one flowchart. The execution of n flowchart segments (n = 1, 2, 3, ...) is initiated by n ARROWs going from a PROCESS symbol (see Figure 27). These segments execute in parallel. Each segment is executed as if it were the only part of the flowchart currently executing. A segment's execution is terminated with the execution of the END primitive (see Figure 28) or by returning control to the "calling" flowchart. The execution of one segment waits on the completion of the other segment's execution with the WAIT primitive (see Figure 29). If there are n ARROWs to a symbol containing the WAIT primitive, then WAIT is equivalent to END for the first n-1 times that it is executed, but on the nth execution, WAIT is equivalent to a "no-op". Only one of the parallel executing segments returns control to the "calling" flowchart and control is not returned until the execution of all segments is terminated. These conventions are analogous to the FORK and JOIN of Conway[4] for conventional languages.

The availability of data for flowchart segments executing in parallel is supervised by the flowchart executor. For example, there is the problem of insuring that data required on one segment's execution is defined and available for use.

An extension of the element type designations solves this problem. For each existing element type, a conditional version of this element type is added to the set of allowed types. For example, conditional floating point number is an additional type corresponding to floating point number. The conditional version of an element type has all the properties of the original and has the additional property of being "defined" or "undefined". The storing of a value into a

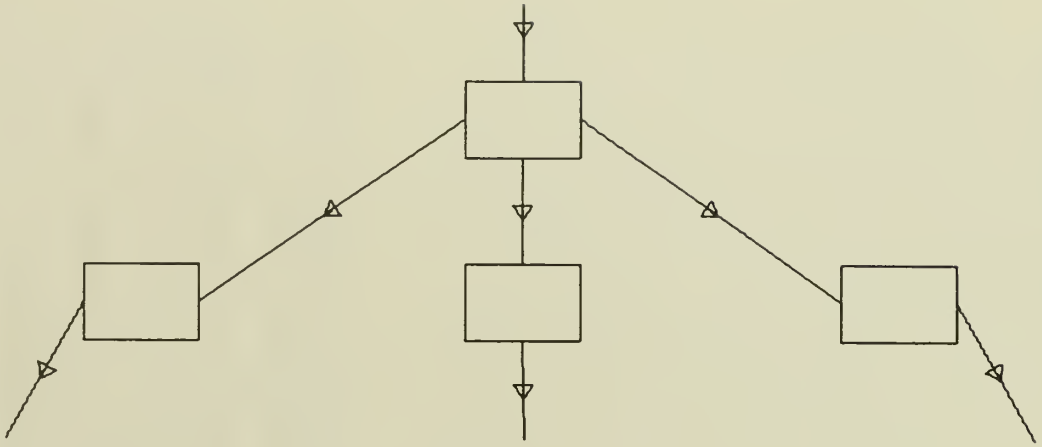


Figure 27. Initiation of the Parallel Execution of Three Flowchart Segments

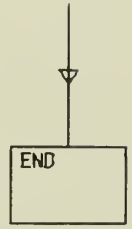



Figure 28. Termination of a Flowchart Segment's Execution

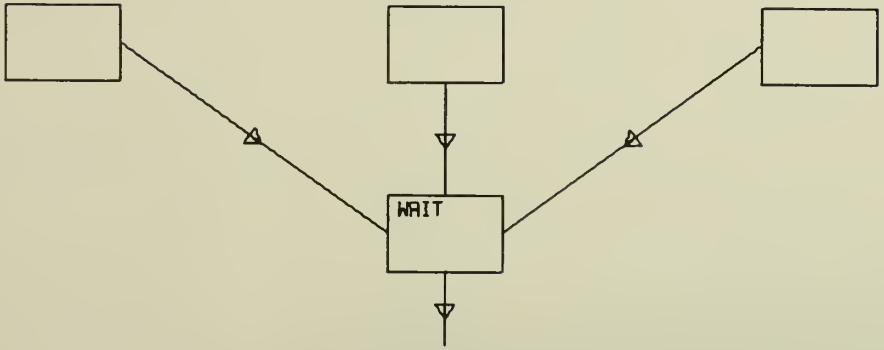



Figure 29. One Flowchart Segment Waiting on the Completion of the Execution of Two Other Flowchart Segments

conditional element makes the element "defined". The fetch of information from a "defined" conditional element provides the type and value of the element and makes the element "undefined". The fetch of information from an "undefined" conditional element suspends the flowchart segment's execution. The segment's execution is resumed when the conditional element is "defined" by another flowchart segment. Thus when a segment references an identifier, and the identifier is "undefined", then the segment's execution is suspended until the identifier is "defined".

These conditional elements are analogous to the use of the circle indicators in Sutherland's data flowcharts. One of his symbols with a circle on one of its inputs is only activated for each new value on that input. This has the effect of suspending execution on part of the flowchart until a datum is available. (With these conditional elements or Sutherland's circles, the programmer must specify in advance which data is involved jointly in simultaneously executing flowcharts.) Sutherland's data flowcharts execute "in parallel" depending upon the availability of data. That is, a symbol is executed when its inputs are defined. For this,the only explicit convention regarding the indication of parallelism is the splitting of the data lines.

In an environment with a fixed number of available processors, the control flowchart notation serves to indicate flowchart segments that can be executed in parallel. Segments are executed in parallel when processors are available. Otherwise, they are executed sequentially in a manner determined by their initiations and terminations in the following way: the flowchart's execution proceeds from the entry point until a symbol is reached where n flowchart segments are to be initiated.

If n processors are available, then the execution of the n flowchart segments is initiated. If only m < n processors are available, then m arbitrarily chosen flowchart segments are initiated. The execution of the n - m other flowchart segments is retained in a suspended state by the flowchart executor. If the execution of a flowchart segment is suspended by an "undefined" conditional element or is terminated by either the END or WAIT primitive, then the execution of one of these n - m other flowchart segments is initiated.

In the special case where only one processor is available, this procedure reduces a parallel flowchart into a sequential flowchart. This allows a user to write his program as a set parallel process and the flowchart executor does the serialization of the program.

## 6.10 Flowchart Recursion

An FPL/I flowchart program can invoke itself recursively. For example, the FPL/I flowchart FACT in Figure 30 decides at (1) if the input argument is greater than one. If it is, then FACT is called recursively at (2) with the original input argument reduced by 1. (The flowchart INTGI of Figure 15 is also recursive.) There is no distinction made between a flowchart calling itself recursively and a flowchart calling any other flowchart. Any flowchart is called by placing its name in a symbol's text string.

## 6.11 FPL/I Implementation

Flowchart execution is accomplished with an interpreter operating on the supporting computer. The advantages offered by interpretation are: 1. the language is independent of a particular computer hardware system, 2. the source language is available for providing

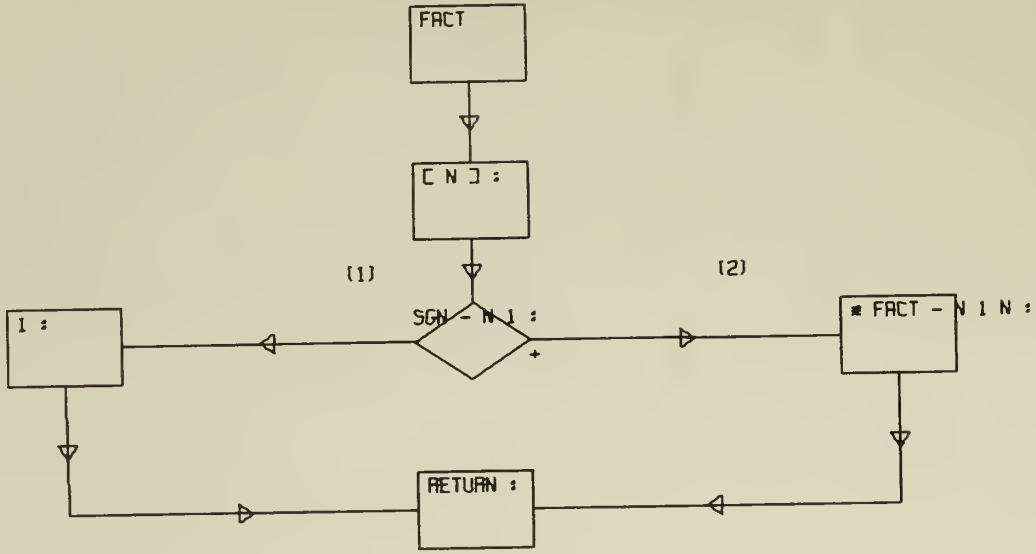



Figure 30. FPL/I Factorial Program FACT

debugging information, and 3. the language is more easily implemented and changed than if implemented by a compiler. Interpretation has the disadvantages that it is slow and requires extra memory space.

FPL/I's pointer was initially provided as a means of bootstrapping FPL/I's implementation. The strategy was to implement the flowchart executor/debugger with a basic set of pointer primitives and then to implement FPL/I's arithmetic functions as flowcharts. However, the number of basic pointer primitives required outnumber by at least a factor of two FPL/I's arithmetic functions. For this reason, FPL/I's arithmetic functions were implemented as primitives. However, the pointer and some pointer primitives were implemented to provide tools used in debugging the FPL/I interpreter. (For example, see the LIST flowchart of Figure 18, Section 5.12.)

## 6.12 Flowcharts and Conventional Languages

The methods of automatically generating a flowchart from a conventional program are more valuable in CAPS than they are in conventional systems. For example, a program's flowchart can be generated by the supporting computer and transmitted to the display device for display on the CRT. If the programmer does not like the flowchart's layout, then he can quickly alter it with the light pen. The supporting computer retains these alterations for use in subsequent flowchart generations from the same program.

In the opposite direction, there are uses for converting flowcharts into conventional programs. For example, this conversion allows flowcharts to be a programming media for existing programming languages. A result of converting from a flowchart into a conventional

program is indicated in Figures 31 and 32. A program to translate flowcharts into conventional language programs is being developed as a continuation of this flowchart investigation.

This conversion program requires a description of the conventional language and then, based upon this description, it converts flowcharts into conventional programs. The conventional program is represented as a set of card images. The language description specifies the formats for labeled, conditional, transfer, unconditional transfer, return, and termination statements. Also specified is the character string used to indicate the scope of the iteration sequence. Figure 33 gives the description of FORTRAN used in converting the flowchart of Figure 31 into a conventional program of Figure 32.

The conversion strategy is to sequence through the flowchart symbols, converting the text in each symbol into a set of card images. A labeled statement is produced as the first card image resulting from each symbol that has two or more ARROWs coming to it or that has an ARROW coming to it from a DECISION symbol. This label is used for indicating control transfers to the symbol. The first three characters of the label are the same for the entire flowchart and are input as part of the conventional language description. The remaining three characters of the label represent the symbol's internally defined number. A symbol without ARROWs coming to it is the first symbol processed in the conversion and processing continues along the paths indicated by the ARROW symbols. The TEXT from a PROCESS symbol is converted directly into card images if no more than one ARROW goes from the symbol. If two ARROWs go from a PROCESS symbol, then the last card image in the PROCESS symbol is an iteration statement (DO in the case of FORTRAN) and

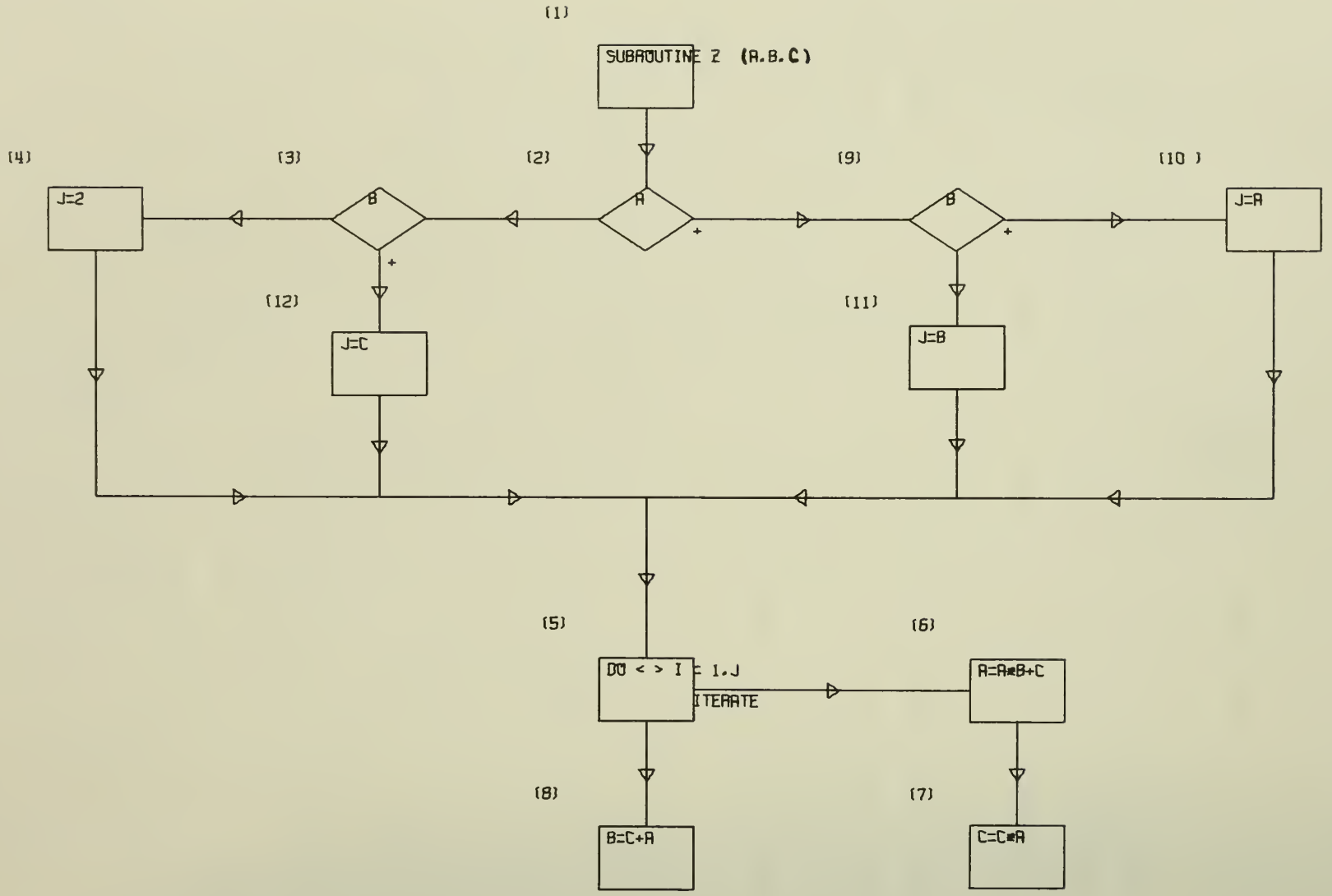



Figure 31. Flowchart Z Written in "Flowchart FORTRAN"

```
        SUBROUTINE  Z(A,B,C)                    (1)
        IF ( A )    1003, 1003, 1009            (2)
1003    IF ( B )    1004, 1004, 1012            (3)
1004    J = 2                                   (4)
1005    DO  1999    I = 1,J                     (5)
        A = A * B + C                           (6)
        C = C * A                               (7)
1999    CONTINUE
        B = C + A                               (8)
        RETURN
1009    IF ( B )    1011, 1011, 1010            (9)
1010    J = A                                   (10)
        GO TO 1005
1011    J = B                                   (11)
        GO TO 1005
1012    J = C                                   (12)
        GO TO 1005
        END
```

Figure 32. FORTRAN Subroutine Z Converted from Flowchart

The character b indicates a necessary blank character in a card line.

Statement Label:

```
     <bblb>bCONTINUE
```

Conditional Transfer:

```
     bbbbbbIF ( << >> ) < >,< >,<+>

     bbbbbbIF ( << >> ) < >,<0>,< >

     bbbbbbIF ( << >> ) <->,< >,< >

     bbbbbbIF ( << >> ) < >,<0>,<+>

     bbbbbbIF ( << >> ) <->,< >,<+>

     bbbbbbIF ( << >> ) <->,<0>,< >

     bbbbbbIF ( << >> ) <->,<0>,<+>
```

Unconditional Transfer:

```
     bbbbbbGO TO < >
```

Scope of Iteration Indication:

```
     ITERATE
```

Return Statement:

```
     bbbbbbRETURN
```

Termination Statement:

```
     bbbbbbEND
```

Figure 33. A Description of FORTRAN for the Conversion Program

the text on one of the "from" ARROWs indicates the scope of this iteration (see (5) in Figure 31). If the text input as the iteration scope indicator matches the text on an ARROW that goes from this PROCESS symbol, then that ARROW goes to the first symbol in the iteration sequence. If there is no match, then an error condition exists. The iteration statement must have the label of the last statement in the iteration sequence inserted into the place indicated by the "<" and ">" characters. For this purpose, a labeled "NO-OP" statement (CONTINUE in FORTRAN) is created and inserted as the last card image generated from the last symbol in the iteration sequence. (There must be exactly one last symbol in the iteration sequence.) The label on this labeled statement is generated with the symbol number of a nonexistant symbol.

The text in a DECISION symbol is inserted into a card image derived from one of the conditional statement forms (see (2), (3), and (9) in Figure 31). The DECISION symbol's text is inserted into the position occupied by the "<<" and ">>" characters. The "<<" and ">>" characters are removed from the conditional statement and the characters to the right of the ">>" characters are shifted to the right to allow for the insertion of an arbitrary amount of text. The particular conditional statement form that will contain a DECISION symbol's text is determined by matching the text on the ARROWs from the DECISION symbol with the text contained in the "<" and ">" characters in each conditional statement form. Once a conditional statement form is chosen, its "<" and ">" and contained characters are replaced by the labels of the symbols that are indicated by the ARROWs.

A symbol, that is not in an iteration sequence and that has no ARROWs from it, is a return symbol and has a return statement included

as the last card image that is generated from the symbol's text. After all the symbols have been converted, the termination statement is produced as the last card image for the program.

### 6.13 Symbol Shape Semantics

For flowchart programming languages, the shape of the flowchart symbols may specify all, some, or none of a flowchart's semantics with the symbol's text specifying the remainder of the semantics. For FPL/I with the five CAPS allowed symbol shapes, only a minor portion of a flowchart's semantics are specified by the symbol's shape. The main emphasis is on the symbol's text. In Sutherland's work with an arbitrary number of constructable symbol shapes available, the semantics of a constructed symbol (i.e. a flowchart) are conveyed entirely by the symbol's shape.

What is important to note here is that the restriction to a specific, finite number of symbol shapes limits the flowchart programming language whose semantics are conveyed entirely by the shape of its symbols. For example, a flowchart program for a Turing machine can be composed of 4 distinct symbol shapes and the arrow connections between the symbols. The semantically meaningful text on a symbol is necessary for "higher level" languages when only a finite number of symbol shapes are available.

# CHAPTER VII

# CONCLUSIONS

## 7.1 The Relationship of CAPS and Other Work in This Area

A basic difference between GRAIL, the Sutherland investigation, and the CAPS study is the computer hardware. GRAIL and Sutherland use displays that are connected directly to large, high speed computers. When the displays are in use, the computers are dedicated to the task of manipulating the data and the inputs. CAPS uses a relatively inexpensive remote display device that uses a large, high speed computer in a time sharing environment. The large, high speed computer provides service on a demand basis for the display device. This hardware scheme is desirable for economic reasons. The cost of available display devices is going down and the cost of the large computer is directly related to the actual use that is made of it. In this way, CAPS is an example of a class of systems that can develop in the near future.

The different methods of constructing flowcharts in these three systems reflect differences in hardware, internal flowchart representation, and project emphasis. For example, the only input device for the GRAIL system is the RAND tablet[5]. The flowcharts that are drawn with the stylus are variable sized rectangles, diamonds, circles, and lines. The system recognizes hand written text for the input of textual information. This reflects RAND's emphasis on making the CRT a common working surface between the human and the computer.

The flowchart programming languages of the three systems are also different. The GRAIL and CAPS flowcharts are similar in that their emphasis is on a symbol's text rather than on a symbol's shape. This in

contrast to the data flowcharts of Sutherland where the emphasis is on a symbol's shape.

The GRAIL flowcharts represent assembly language programs for an actual computer while CAPS flowcharts represent programs for a simulated computer with a list oriented structure and memory. Sutherland's flowchart symbols represent operators that are activated by inputs and produce outputs. The debugging tools of the three systems are roughly comparable for the three different flowchart programming languages.

The FPL/I language uses techniques that are similar to those that are used in the SPRINT[16] list processing language. In SPRINT, as in FPL/I, there is a name table that supplies the value associated with an identifier. In the SPRINT instruction stream, the programmer indicates to the processor which "words" are to be executed as instructions and which "words" are to be taken as data. In FPL/I, the identifiers are flagged as instructions or data in the name table rather than in the instruction stream. In the FPL/I instruction stream, an identifier, specified as an instruction in the name table, may temporarily be specified as a datum with the use of the ">" primitive and the "<" delimiter. In FPL/I and SPRINT, lists are used for data manipulation and storage. SPRINT uses one-way linked lists, while FPL/I uses two-way linked lists. The two-way links of FPL/I simplify memory allocation and garbage collection. Each link occupies 13 bits of one 52 bit word in the supporting computer and is not a particularly inefficient use of storage. However, the two-way links are not essential for FPL/I and could be replaced by one-way links.

## 7.2 CAPS Flowchart Editing System

The experimental CAPS allows the construction and editing of flowcharts. Starting with an initial version, the system's operation has been improved with the addition of more operations (GROUP, VERT, HORZ, CHNG, XDELT, and the provision for large flowcharts) and more refined operating techniques (improved light pen tracking and use of the displayed border for symbol positioning). The operation of the system is reasonable and is taught to the new user in a matter of minutes.

The use of the teletype keyboard and light pen as input to the display device requires that the user's attention frequently be shifted from one input device to the other. This is a source of annoyance to the user. The use of a RAND tablet[5] is an attractive alternative to the use of the teletype and light pen. (Soon, the experimental CAPS display device will be equipped with a tablet.) The tablet and stylus perform the same drawing functions that the light pen does. However, for text input, the display device must recognize handwritten text from the tablet or must display a keyboard on the CRT which is "typed on" with the stylus. The recognition of handwritten text may be beyond the display device's capabilities. The supporting computer cannot do the recognition because the uncertain response times of the time sharing system make the handwriting slow, awkward, and tedious.

If the display device cannot do the text recognition, then it must display a keyboard on the CRT and allow the user to "type" with the stylus. For much text input, the speed of this method of "typing" versus that of touch or "two fingered hunt and peck" typing at the teletype keyboard is a comparison that must be resolved through experimentation.

However, the purpose in having the tablet is to reduce the number of input devices to the system and using the keyboard, in addition to the tablet, defeats this purpose.

The flowchart construction system is restricted in that it does not allow for the definition of multilevel flowcharts. These are not allowed because of the slowness of the flowchart representation transmission. An approximation to a multilevel flowchart facility is provided by the flowchart's ability to invoke other flowcharts. The success of this approximation depends on the programmer and his applications.

This system does not provide facilities similar to the drawing constraints of SKETCHPAD. The absence of these constraints is not a serious restriction because the system does provide drafting tools (the alignment and the "group" operations).

For flowchart hardcopy, CAPS produces incremental plotter tapes which are then plotted off-line. (All of the flowchart drawings in this thesis were produced in this fashion.) The length of the plotter turnaround time offers pressure for providing some other hardcopy media such as photographs or printer generated flowcharts.

The flowchart construction system is not a generalized drawing scheme like SKETCHPAD but is a more specialized system for drawing flowcharts and similar drawings. The system is specialized in that predefined symbols and symbol linkages are offered rather than constructable symbols and linkages as in SKETCHPAD. The advantages gained with predefined symbols are that they are easy to add to the flowchart, easy to connect together, and occupy little memory space in the flowchart representation. A disadvantage to using pre-defined symbols is that a

different shaped symbol is difficult to add to the system. For example if an octagonal symbol is needed for some purpose in a flowchart, then it must painstakingly be added to the system. An area for further investigation is the automatic generation of specialized drawing systems similar to the flowchart drawing system. For example, a specialized drawing system for electronic circuits should be able to be automatically generated once the set of allowed symbols is defined in a suitable input language. In the environment of the small display device, these specialized drawing systems are valuable because the full generality of SKETCHPAD cannot be offered due to memory restrictions.

## 7.3 CAPS Flowchart Programming

The use of FPL/I indicates that the CAPS types of flowcharts are valuable as program writing tools. However, languages more powerful than FPL/I need to be developed. For example, the data types of byte, fixed point number, character string, array, pointer, and the necessary primitives are needed. The scope of an identifier may need to be changed to correspond to the scope suggested in Section 6.3 to allow ALGOL compatable, EXTERNAL, and graphically defined identifier scope. In addition, for the convenience of the user, a primitive or flowchart needs to be provided to convert from infix arithmetic notation to the prefix notation used internally. A flowchart programming language with these capabilities would allow application of flowchart programming techniques to many information processing areas which include numeric processing, symbol manipulation, list processing, and operating systems. With this wide range of applicability, more people will be able to use flowcharts as programming media and furnish valuable feedback on the CAPS'

flowchart programming languages. Such feedback would include information on additional debugging aids that are needed above and beyond those developed for FPL/I.

An area for further investigation is the provision for "top-down" approach in a flowchart programming language. The "top-down" programming process starts with a grossly detailed flowchart of the program which is "executed" in order to test its correctness and feasibility. Once this flowchart is "debugged", each symbol of the flowchart is treated as a separate program and a flowchart is written for it. (This is where the multilevel flowchart facility would be useful.) This process continues until a flowchart is written in sufficient detail that a flowchart program can be written for it in an existing programming language. When all of the flowcharts have flowchart programs written for them, then the entire program has been written and debugged at the same time.

This is a brief description of what in time and with development might be a suitable approach to the "programming process". What is needed for this "top-down" procedure are "languages" for the grossly detailed flowcharts and provisions for "executing" and "debugging" these flowcharts.

## 7.4 Summary

In summary, the investigation described here considers problems encountered in allowing individuals to code, execute, debug, and document computer programs in the media of flowcharts. The feasibility of this use of flowcharts is demonstrated by the experimental system constructed in the course of this investigation. The economic feasibility of this

use of flowcharts is indicated by the form of the experimental system hardware and the expected decline in hardware costs.

As programming media, flowcharts must inevitably be compared to card images in time sharing and batch processing systems. The use of the CAPS flowcharts by a small group of enthusiasts is not grounds for a fair comparison; but based upon our experience and observations, flowcharts appear to be more valuable than card images for large complicated programs written by a group of people. The execution trace of a flowchart is valuable not only as a debugging tool, but also as a teaching tool for programming and as "living" program documentation. Card images appear to have the economic edge over flowcharts for simple, short programs. A more rigorous testing of these observations is another area for further investigation.

# REFERENCES


1. Brearly, H. C., "ILLIAC II--A Short Description and Annotated Bibliography," Report No. 173, Department of Computer Science, University of Illinois, February, 1965.

2. Burroughs Corporation, "Burroughs B5500 Information Processing Systems Reference Manual," Burroughs Manual No. 1021326.

3. Klark, W. A., et al., "The Lincoln TX-2 Computer Development," Proceedings of the WESTERN Joint Computer Conference, 1957, Institute of Radio Engineers, pp. 143-171.

4. Conway, M. E., "A Multiprocessor System Design," AFIPS Conference Proceedings, Vol. 24, 1963, Fall Joint Computer Conference, pp. 139-146, Spartan Books, Baltimore, Maryland (1963).

5. Davis, M. R., and Ellis, T. O., "The RAND Tablet: A Man-Machine Communication Device," AFIPS Conference Proceedings, Vol. 26, 1964 Fall Joint Computer Conference, pp. 325-331, Spartan Books, Baltimore, Maryland (1964).

6. Digital Equipment Corporation, "PDP-7 Users Handbook," DEC Manual F-75.

7. Digital Equipment Corporation, "PDP-8, Programmed Buffered Display 338 Programming Manual," DEC Manual DEC-08-G61C-D.

8. Digital Equipment Corporation, "PDP-8 Users Handbook," DEC Manual F-85.

9. Ellis, T. O., and Sibley, W. L., "On the Problem of Directness in Computer Graphics," in Emerging Concepts in Computer Graphics, Benjamin Press (1968).

10. Gear, C. W., et al., "A Status Report on the Design and Construction of a Low Cost Graphical Terminal," File No. 742, Department of Computer Science, January, 1968.

11. Haibt, L. M., "A Program to Draw Multilevel Flow Charts," AFIPS, Proceedings of the WESTERN Joint Computer Conference, 1959, Institute of Radio Engineering, pp. 131-136.

12. Hain, G. and Hain, K., "Automatic Flowchart Design," Proceedings 1965 ACM National Conference, ACM Publications, pp. 513-523.

13. Illiffe, J. K., "The Use of the GENIE System in Numerical Calculations," Annual Review of Automatic Coding, Vol. II, pp. 1-28, Pergamon Press (1961).

14. International Business Machines Corporation, "IBM 7090/7094 Programming Systems FORTRAN II Programming," IBM Manual No. C28-6054-4.

15. International Business Machines Corporation, "IBM Operating System PL/I Language Specifications," IBM Manual No. C28-6571-4.

16. Kapps, C. A., "SPRINT: A Direct Approach to List Processing Languages," AFIPS Conference Proceedings, Vol. 30, 1967 Spring Joint Computer Conference, pp. 677-683, Thompson (1967).

17. Knuth, D. E., "Computer-Drawn Flowcharts," Communications of the ACM, Vol. 6, No. 9, pp. 555-563, September, 1963.

18. Naur, Peter (Editor), "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM, Vol. 6, No. 1, pp. 1-17, January, 1963.

19. Roberts, L. G., "Graphical Communication and Control Languages," Proceedings of the Second Congress on Information System Sciences, pp. 211-217, Spartan Books, Baltimore, Maryland, (1964).

20. Sherman, Phillip M., "FLOWTRACE, A Computer Program for Flowcharting Programs," Communications of the ACM, Vol. 9, No. 12, pp. 845-854, December, 1966.

21. Sutherland, I. E., "SKETCHPAD: A Man-Machine Graphical Communications System," Technical Report No. 296, Lincoln Laboratories, MIT, January, 1963.

22. Sutherland, W. R., "On-Line Graphical Specification of Computer Procedures," ESD-TR-66-211, Lincoln Laboratories, MIT, May, 1966.

# APPENDIX A

# AN EXPERIMENTAL CAPS COMPUTER SYSTEM

A CAPS computer system consists of a display device, a supporting computer, and a communication link between the display device and the supporting computer. An experimental CAPS computer system has been assembled in the Department of Computer Science, University of Illinois. In this experimental system, the display device is a Programmed Buffered Display type 338, the supporting computer is the ILLIAC II, and the communication link between the two computers is a direct connection, similar to a leased phone line, that has a speed of 100 characters per second.

## A.1 The Display Device

The display device, the commercially available Programmed Buffered Display type 338 (PBD-338), is manufactured by the Digital Equipment Corporation. Functionally, the PBD-338 is divided into a small, self contained computer and an attached display processor. The small computer provides I/O with the user and supporting computer and does display maintenance. The display processor generates point and line drawings on the CRT. The small computer's memory is used for display image storage.

The small computer for the PBD-338 is the Programmed Data Processor-8 (PDP-8). The PDP-8 has memory, arithmetic, and I/O facilities. In the experimental CAPS computer system, the PDP-8's main memory is 16,384 words of 12 bit 1.5 μsec memory. Secondary storage is magnetic tape (DECtape) and punched paper tape. For secondary storage access, the PDP-8 has two DECtape transports with an average 12 bit word

transfer rate of 8,750 words per second and a 10 character per second paper tape reader and punch. The PDP-8 has a 10 character per second teletype for user communication and a 100 character per second remote port for supporting computer communication. Other PDP-8 facilities include 1. arithmetic, logical and I/O registers (AC and L), 2. an interval timer, 3. interrupt capability, and 4. facilities for communication with the display processor. For a more detailed description of the PDP-8 see [7].

The display processor is a specialized computer that generates point and line drawings on the attached CRT. The processor's program resides in the PDP-8's memory, and display instructions are fetched from the memory by the display processor on a demand basis. These memory accesses are transparent to the PDP-8's executing program. The display processor's program sequencing and program control transfers are done by the display processor rather than by the PDP-8.

In order to produce the point and line drawings, the display processor has program addressable registers whose values represent the CRT's beam coordinates, beam intensity, drawing mode, and drawing scale. Under program control, the display processor can change the values of any of these registers and, as a result, generate the drawings on the CRT.

For output to the user, the display processor uses the CRT and an array of 12 indicator lamps. The indicator lamps are another of the display processor's program addressable registers. For input to the display processor, the user has a light pen and 12 push buttons. Each button complements the state of an associated output indicator lamp when pushed. The light penning of a displayed point inputs the CRT's beam

X-Y coordinates which are manipulated by the display processor. For example, the display processor detects pen hits on light buttons and tracks the light pen. When more complicated input data manipulations are required, the PDP-8's executing program is interrupted, the input information is transmitted to the PDP-8, is processed by the PDP-8, and then is returned to the display processor in the form of program changes. For a more detailed description of the display processor and the PBD-338 see [8].

## A.2 The Supporting Computer

The supporting computer for the experimental system is the ILLIAC II. ILLIAC II is a high speed digital computer designed, assembled, and maintained by the Department of Computer Science, University of Illinois.

The memories for the ILLIAC II include:

1. 10 words of 0.2 μsec transistor memory
2. 8,192 words of 1.8 μsec core memory
3. 65,536 words of drum memory, 8.5 msec average access time, 7.8 μsec word period
4. magnetic tapes and disk files

In 1., 2. and 3., a word contains 52 bits of information. For arithmetic, there is one floating point register and 16 fixed point registers. The single precision floating point operands are 52 bits long, with a 45 bit fraction. The fixed point operands are 13 bits long. The fixed point registers also serve as index registers. Some approximate operation times are as follows:

| | |
|---|---|
| Floating point add or subtract | 2.5 to 3.5 μsec |
| Floating point multiply | 6.3 μsec |
| Floating point divide | 16.0 μsec |
| Fixed point add or subtract | 2.0 μsec |
| Fixed point indexing | 1.0 μsec |

Floating point operations and fixed point operations are done in parallel under program control in order to reduce the effective operation time. For example, one floating point addition and two fixed point additions are done in parallel in 4.0 μsec, while the same operations require 6.5 μsec if they are done sequentially. High speed I/O is controlled through other registers and is done concurrently with arithmetic.

Time sharing console (teletype) I/O is controlled by a satellite processor, the Programmed Data Processor-7 (PDP-7) made by the Digital Equipment Corporation. The PDP-7 is attached to the ILLIAC II via a high speed data channel. The basic unit of communication between the ILLIAC II and the PDP-7 is a teletype line. The PDP-7 communicates with each attached teletype on a character by character basis. The PDP-7 is considered an integral part of the ILLIAC II I/O facilities. For a more detailed description of the ILLIAC II and the PDP-7 see [1] and [6].

## A.3 The Communication Link

The PBD-338 is connected to the PDP-7/ILLIAC II as a 100 character per second teletype. The connection is a twisted pair of wires that is replacable by a dial up telephone connection.

# APPENDIX B

# THE CAPS FLOWCHART EDITING SYSTEM

A description of the experimental version of the flowchart editing system is included in this appendix.

## B.1 Flowchart Display

A set of words is displayed on the CRT (see Figure 34). These words are names of the system's operations and are used as light buttons for the control of these operations. The light penning of a light button turns the current operation off and turns on the indicated operation. If the light button for the current operation is light penned, then the operation is turned off. The symbols displayed at the bottom of the CRT are symbol light buttons. They are used in adding symbols to the flowchart. The displayed border is used in moving and aligning symbols.

## B.2 Basic Operations

To MOVE a PROCESS, DECISION, TEXT, or CONNECTOR symbol on the flowchart, turn on the MOVE operation and light pen a symbol. A tracking cross indicates the pen's position on the display after a symbol has been light penned. The tracking cross and symbol move with the light pen. The symbol's position is fixed and the tracking cross is removed by closing the pen's shutter.

The light penning of the displayed border on the CRT moves the symbol selected for the MOVE operation. The light penning of a vertical border moves the symbol vertically until it is horizontal with the light pen. The light penning of a horizontal border moves the symbol horizontally until it is vertical with the light pen.

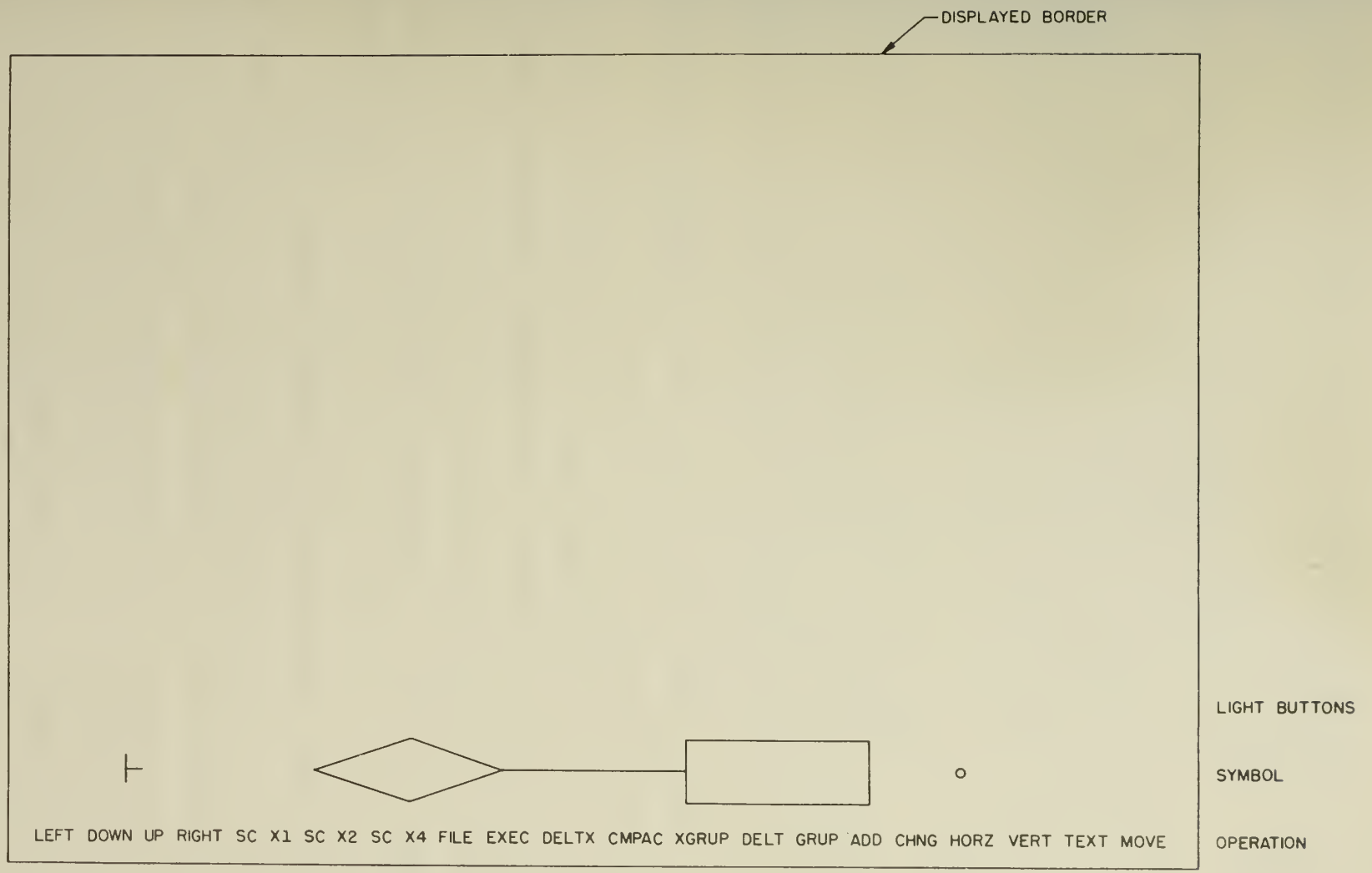



FIGURE 30
FLOWCHART EDITING DISPLAY

To MOVE an ARROW symbol, move the ARROW's "to" or "from" symbol.

To ADD a PROCESS, DECISION, TEXT, or CONNECTOR symbol to the flowchart, turn on the MOVE operation and light pen a symbol light button. This adds a new symbol to the flowchart near the symbol light button. The type of the new symbol is determined by the symbol light button that is selected. Once a symbol has been added to the flowchart, it is manipulated in the same way that any other symbol is manipulated.

To ADD an ARROW symbol to the flowchart, turn on the MOVE operation and light pen the ARROW symbol light button. This causes the word "FROM" to be displayed which indicates that the ARROW will come from the next light penned symbol. When the "from" symbol is selected, "TO" replaces "FROM" which indicates that the ARROW will go to the next light penned symbol.

If the ARROW does not come from a DECISION symbol, then it is displayed immediately when the "to" symbol is light penned. If the ARROW does come from a DECISION symbol, then the text editor is entered when the "to" symbol is light penned. This allows text to be edited onto the ARROW. The ARROW is displayed when the text editor operation is terminated. After the initial ARROW is displayed, a sequence of ARROWs is added by light penning a sequence of "to" symbols. Each of these ARROWs comes from the previous ARROW's "to" symbol. The MOVE operation is turned off and then on again to add another symbol.

The TEXT operation is used to edit text onto a symbol. When the TEXT operation is on and a symbol is selected with the light pen, the flowchart display is replaced by the text editor display. The text editor display consists of a page of text, a text cursor, and a RETURN

light button. The text page (32 lines per page, 64 characters per line) contains the text from the light penned symbol. The position of the text cursor is controlled by the light pen or from the keyboard. The text is edited relative to the text cursor.

A symbol's text editing operation is terminated with the light penning of the RETURN light button. This causes the text editor display to be replaced by the flowchart display with the edited text appearing in its symbol on the flowchart. Another symbol is selected for text editing with the light pen.

To delete a symbol from the flowchart, turn on the DELT operation. The next light penned symbol is deleted. Also, any connected ARROWs are deleted. The DELT operation is turned off after one deletion.

## B.3 Extended Operations

A flowchart has a maximum size of 40" x 40". The CRT is a 10" x 10" window through which the large flowchart is viewed. The operations UP, DOWN, LEFT, and RIGHT move the CRT window over the flowchart in the indicated directions. The flowchart is projected onto a torrus for the purpose of moving the window and therefore, no flowchart edges are encountered.

For viewing the flowchart, the scale operations SC X1, SC X2, and SC X4 are provided. One of these operations is always on. The operations SC X1, SC X2, and SC X4 display 40" x 40", 20" x 20", and 10" x 10" portions of the flowchart, respectively. In SC X1 and SC X2 the symbols are displayed at reduced sizes and without text. In SC X4, the symbols are displayed at full size with text. All editing operations are possible in any scale operation.

The GROUP operation allows the user to divide a flowchart's symbols into sets of symbols that are and are not members of "the group". The group membership affects editing operations. For example, if a group member is MOVEd, then every member of "the group" is MOVEd as if all the "group" members are connected by a rigid structure. If a "group" member is deleted, then every member of the "group" is deleted.

When the GROUP operation is turned on, the set of symbols that are members of "the group" are indicated by their blinking displays. A light penned symbol is moved from its current membership set into the other set. The symbols do not blink when the GROUP operation is turned off.

The XGRUP operation takes all of the symbols in "the group" and puts them into the set of symbols that are not in "the group". This operation clears "the group" membership in order that a new "group" can be formed.

The ADD operation is used to duplicate a symbol or "the group" of symbols. When the ADD operation is on, a duplicate copy, text included, of the light penned symbol is added to the flowchart. If the symbol is a member of "the group" then a duplicate of "the group" is added to the flowchart. "The group" membership is changed when it is duplicated. The original "group" members are taken from "the group" and the duplicates are inserted into "the group".

The operations VERT (VERTical) or HORZ (HORiZontal) align selected symbols on vertical or horizontal lines. When the VERT operation is turned on, the word "START" is displayed on the CRT indicating that a reference point is to be chosen. A reference point is any position on the border or any symbol. The word "START" is removed from the

display when the reference point is light penned. While the VERT operation is on, any light penned symbol is MOVEd horizontally until it is on a vertical line with the reference point. If a light penned symbol is a member of "the group" then "the group" is MOVEd horizontally until the selected symbol is on a vertical line with the reference point.

The HORZ operation works in a manner similar to the VERT operation.

The CHNG operation allows a symbol's type to be changed without deleting the symbol, its text, and its connections. When the CHNG operation is turned on, the type of the first light penned symbol is changed into the type of the second light penned symbol. The CHNG operation is turned off after one change is performed. If the first light penned symbol is a member of "the group" then each member of "the group" has its symbol type changed.

The DELT operation is extended to allow recall of previous deletions. The DELT operation removes the indicated symbols from the display but not from the internal flowchart representation.

The DELTX operation returns the most recently deleted symbol or set of symbols to the flowchart.

The CMPAC (CoMPACt) operation removes the deleted symbols from the internal flowchart representation. The DELTX operation cannot return deleted symbols to the flowchart once the CMPAC operation has been performed.

The FILE operation is used for local flowchart storage. The EXEC operation provides for communication with the supporting computer.

# APPENDIX C

# FPL/I PRIMITIVES

This is a description of the implemented set of FPL/I primitives, their input/output arguments, and their actions on these arguments. The brackets, "{" and "}", enclose each required input argument for a primitive. The name enclosed in the brackets is a description of the properties that the argument must have. If an actual input argument does not fit this description, then an error condition exists.

## C.1 Arithmetic Primitives

Syntax:

+ {element} {element}

- {element} {element}

* {element} {element}

/ {element} {element}

1st 2nd arguments

Semantics

The execution of an arithmetic primitive combines the two input arguments into one output argument in the following way:

| Primitive | Result |
|---|---|
| + | {element} + {element} |
| - | {element} - {element} |
| * | {element} * {element} |
| / | {element} / {element} |

1st 2nd arguments

If both arguments are floating point number elements, then these operations are performed. If one of the arguments is not a

floating point number, then an error condition exists.

Examples:

```
* + 3.4 5.6 2.9 :

/ - / X Y X 2 :
```

C.2 Name Table Manipulating Primitives

Syntax:

SET {identifier name} {type name} {value}

FREE {element}

= {element} {element}

1st 2nd 3rd arguments

{identifier name} ::= Not more than six alphanumeric characters. The first character must be a letter.

{type name} ::= FLOAT | FLOWCH | PRIMIT

{value} ::= a floating point number, a flowchart, or a primitive depending on the corresponding type declaration.

Semantics:

The SET primitive adds a new element to the top of the name table. This element's name, type, and value are provided by the {identifier name}, {type name}, and {value} arguments.

The FREE primitive removes the first occurrence of the argument's name from the name table.

The = primitive stores the type and value of the 2nd argument into the entry in the name table for the name of the 1st argument. If there is no name table entry for the name of the 1st argument then an entry is created. If the 1st argument does not have a name then an error condition exists.

Examples:

```
SET A FLOAT 3 :

SET B FLOAT + A 6 :

SET PLUS PRIMIT <+> :

SET SQRT1 FLOWCH <SQRT> :

FREE A :

= B * + A B B :
```

## C.3 Condition Register Setting Primitives for Arithmetic Conditions

Syntax:

```
SGN {element}

SIGNUM {element}

OVFLOW
```

Semantics

The SGN and SIGNUM primitives store "+", "0", "-" in the condition register if the argument is a positive, zero, or negative floating point number. If the argument is not a floating point number, then an error condition exists. The SGN primitive tests the value of the argument. The SIGNUM primitive takes the name of the argument and tests the value of the name table entry for that name.

The OVFLOW primitive stores "YES" or "NO" in the condition register if the last arithmetic primitive did or did not cause an overflow condition.

Examples:

```
SGN * * A B C :

SGN + A B :

SIGNUM A :

OVFLOW * * * A A A A :
```

C.4 Input/Output Primitives

Syntax:

PRINT {element}

READ {element}

READER

Semantics:

The PRINT primitive prints out the argument's name, followed by an equal sign (=), and followed by the argument's value. The print out is on the teletype printer or on the CRT.

The READ primitive prints out the argument's name, followed by "=?", and followed by an input request on the teletype keyboard. The text that is received as a result of the input request is appended to the right of the text string

= {element}

and the resulting text string is executed. The execution is done in the same way that a symbol's text is executed. (The input argument for the = primitive is the same as the input argument for the READ primitive.) The execution of this string stores into the name table a new value for the argument of the READ primitive.

The READER primitive gives an input request on the teletype keyboard. The text string that is received as a result of this input request is executed in the same way that a symbol's text is executed. The READER primitive allows text strings, (parameters, operators, and data) to be input as part of the execution of a program.

Examples:

Encountered in a flowchart -

READ A : PRINT A :

Results on the teletype -

```
A      = ?
- * 4 7 * 3 12 :
A      = -8.0000
```

C.5 Input Parameter Manipulating Primitive

Syntax:

```
[{identifier name list}]
{identifier name list}::= {identifier name} |
                          {identifier name list} {identifier name}
RETURN
```

Semantics:

The "]" primitive inserts a new identifier into the name table for each name in the list. The type and value of each identifier is determined by the corresponding input parameter.

The "[" character is a delimiter for the "]" primitive.

The RETURN primitive FREEs the identifiers created by the "]" primitive.

Examples:

The "calling" sequence:

```
X 6 + 9 8 3 :
```

in flowchart X at (1) (see Figure 31), the identifiers A,B, and C are inserted into the name table as floating point numbers with values 6, 17, and 3, respectively. At (2) in X, the identifiers A,B, and C are FREEed from the name table.

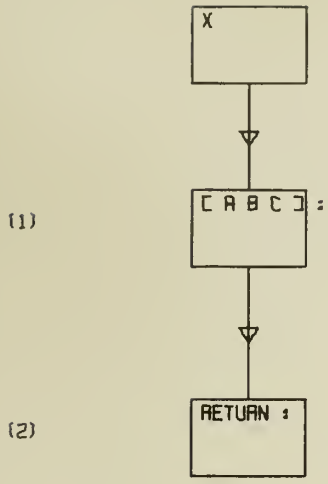



Figure 35. FPL/I Flowchart X

C.6 Execution Scan Manipulating Primitive

Syntax:

< {element list} >

{element list}::= {element} | {element list} {element}

Semantics:

The ">" primitive causes the execution scan of the text string to skip text until the corresponding "<" is encountered. The ">" primitive removes the "<" bracket so that on subsequent scans, the contained text strings can be executed.

Examples:

< A B SQRT > :

C.7 Debugging Primitives

Syntax:

MTRACE {element}

NOTRAC

TRACE

TRACED

NTRACD

QTRACE

NQTRAC

SETIME {element}

INITIA

Semantics:

The MTRACE primitive causes the "execution time" trace of the flowchart that is named by the argument. For this trace, the indicated flowchart is displayed on the CRT and the display of each symbol is blinked as the symbol's text is executed. The trace is activated each time that the flowchart is called. (The experimental system currently allows only one flowchart to be traced at a time.) Within a flowchart whose execution is being traced, the primitives - NOTRAC, TRACE, TRACED, NTRACD, QTRACE, NQTRAC, and SETIME - control the manner in which the tracing is done. These primitives are encountered in the execution of the flowchart.

The NOTRAC primitive suspends the tracing of the flowchart's execution. The flowchart execution continues without the blinking of any of its symbols.

The TRACE primitive resumes the tracing of the flowchart's execution. The use of the NOTRAC and TRACE primitives allow the tracing

of selected positions of the flowchart.

While in the suspended tracing mode, created by executing the NOTRAC primitive, the execution of the TRACED primitive causes only the flowchart's decisions to be traced.

The NTRACD primitive terminates this decision tracing.

The execution of the QTRACE (query trace) primitive causes the flowchart's execution to be suspended after each symbol and an input request to be sent to the teletype keyboard. If the input line for this request is blank, then the flowchart's execution is resumed. If the line is non-blank, then it is executed as if it were text from a symbol. After this line is executed, another input request is given. The execution of these input lines allows the user to specify a particular action. For example, identifier values may be printed out and changed with the PRINT and = primitives, respectively.

The execution of the NQTRAC primitive terminates this mode of tracing.

The SETIME primitive specifies that in the trace of a flowchart, at least z seconds are to be taken in each symbol's execution. The number z is given by the argument for SETIME. This primitive allows the user to trace his flowchart's execution in a slowed down fashion rather than in the faster, real time execution.

The execution of any flowchart is terminated and the system is returned to an initial state when the INITIA primitive is executed.

Examples:

```
MTRACE <SQRT> :

MTRACE <INTG> :

SETIME 1 :

SETIME  / 1 2 ;
```

C.8 Pointer Manipulating Primitives

Syntax:

PRIGHT

PLEFT

PUP

PDOWN

PSHOW

PSHOWX

PSETC {element}

PDELET

Semantics:

The PRIGHT and PLEFT primitives move the pointer right and left one element in the list, respectively.

The PUP and PDOWN primitives move the pointer up and down one element in the list, respectively. These elements with vertical links are internally created for use in the name table and the list representations for flowcharts. If the "pointed to" element does not have vertical links, then an error condition exists.

The PSHOW and PSHOWX primitives print out the "pointed to" element. The PSHOW primitive prints out the element's name and value. The PSHOW primitive prints out the element's representation in an octal format.

The PSETC primitive "points" the pointer to the argument. Thus the pointer "points" to an element in the processing list. The pointer may be moved right or left. If the "pointed to" element has vertical links, then the pointer may be moved up or down an element in the lists. The names of the internal lists - NAMET (name table), ARITHS

(processing list), and CC (control counter for executing flowcharts) - are internally defined elements that have down links to the appropriate lists. The name of a flowchart is an internally defined element that has a down link to the list that is the internal representation of the flowchart.

The PDELET primitive deletes the "pointed to" element from its list. After the deletion, the element is still "pointed to" by the pointer; but the element is no longer in its list. When the pointer is moved, this deleted element is no longer available.

C.9 Condition Register Setting Primitives for Pointer Conditions

Syntax:

PSGN

PSIGNM

PREOL

PLEOL

Semantics:

The PSGN and PSIGNM primitives store "+", "0", or "-" in the condition register if the "pointed to" element is a positive, zero, or negative floating point number. If the element is not a floating point number, then an error condition exists. The PSGN primitive tests the value of the element. The PSIGNM primitive takes the name of the "pointed to" element and tests the value of the name table entry for that name.

The PREOL and PLEOL primitives store "YES" or "NO" if the "pointed to" element is or is not at the right or left hand end of the list, respectively.

# VITA

Fontaine Kelleam Richardson was born in Fayetteville, Arkansas, on April 26, 1941. In 1963, he graduated with the degree of Bachelor of Science in Mathematics from the University of Arkansas, Fayetteville, Arkansas. In 1964, he graduated with the degree of Master of Science in Mathematics from the University of Arkansas. In 1963 and 1964, he was a member of the computer programming staff at the Computing Center, University of Arkansas. As a graduate student at the University of Arkansas, he was a teaching assistant in the Department of Mathematics. In September 1964, he began his graduate study at the University of Illinois. Since that time he has been a research assistant in the Department of Computer Science.

He is a member of the Association for Computing Machinery.